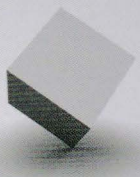


Ressourcesinformatiques

Version en ligne
OFFERTE !
pendant 1 an

Algorithmique

Techniques fondamentales de programmation

Exemples en Python
(nombreux exercices corrigés)

Nouvelle édition

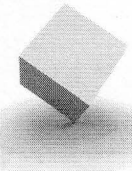
En téléchargement

code source des exemples

BTS, DUT Informatique

Ludivine CREPIN

eni



Ressourcesinformatiques

Algorithmique

Techniques fondamentales de programmation

Exemples en Python

(nombreux exercices corrigés)

Nouvelle édition

Avant-propos

Cet ouvrage s'adresse à toute personne souhaitant commencer à développer des programmes en informatique. Avant de pouvoir écrire du code, il faut d'abord apprendre à réfléchir pour que la machine vous comprenne et exécute vraiment ce que vous lui demandez.

Ce livre vous permet d'acquérir les bases fondamentales de cette réflexion avec l'apprentissage de l'algorithmie, la pensée informatique. Vous apprendrez à manipuler les notions essentielles de la programmation telles que les variables, les opérateurs, les structures, les itérations, les conditionnelles, les instructions, les tableaux et les sous-programmes.

À chaque étape, vous commencez par apprendre à penser avec l'algorithmie pour ensuite appliquer vos algorithmes avec le langage Python. Ainsi, vous avez une preuve concrète de la validité de votre logique, de vos algorithmes et de vos scripts.

Nous commençons par décrire brièvement le fonctionnement de la machine avant de nous lancer dans nos premiers algorithmes et programmes. Chaque chapitre est dédié à une seule notion importante de l'algorithmie avec son équivalent en Python. Nous terminons également ce livre par une introduction à la programmation orientée objet pour que le lecteur puisse continuer sur la lancée et aller plus loin dans le développement informatique.

Pour chaque chapitre, vous pouvez tester vos connaissances avec un quiz corrigé et vous pouvez mettre en pratique avec de nombreux exercices corrigés.

Ce livre est principalement dédié aux lycéens qui suivent l'option de spécialité numérique et sciences informatiques et aux étudiants qui commencent leur formation en informatique, entrant donc en première année de Licence, DUT ou BTS informatique. Cependant, il n'est pas nécessaire d'étudier l'informatique pour apprendre le développement. Ce livre est donc ouvert à toute personne étant à l'aise avec l'utilisation d'un ordinateur.

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **RIALGPYT** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1
Introduction à l'algorithmique

Chapitre 2
Les variables et opérateurs

Chapitre 3
Conditions, tests et booléens

Chapitre 4
Les boucles

Chapitre 5
Les tableaux et structures

Chapitre 6
Les sous-programmes

Chapitre 7
Passons en mode confirmé

Chapitre 9
Commencer avec l'objet

Chapitre 1
Introduction à l'algorithmique

1. Les fondements de l'informatique

1.1 Architecture de l'ordinateur

L'idée d'une machine pouvant faire plusieurs calculs vient de la **machine de Turing**. Alan Turing est le mathématicien qui, avec son équipe, a réussi à casser le chiffrement de la machine Enigma pendant la Seconde Guerre mondiale, ce qui a donné une énorme avance stratégique et tactique pour vaincre les Allemands. C'est en 1936 qu'Alan Turing eut cette idée de génie.

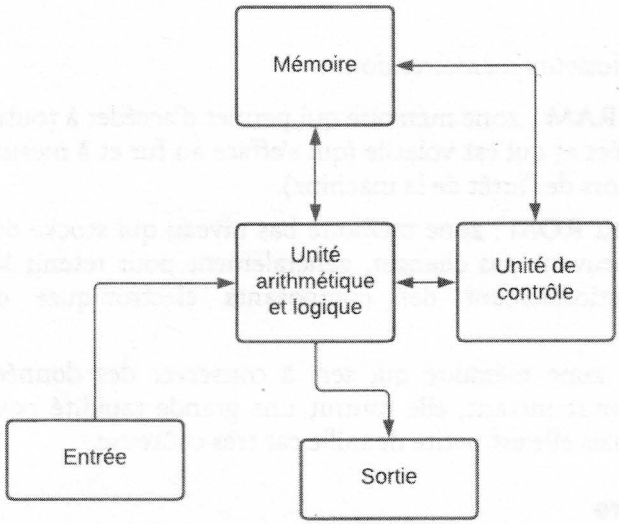
Jusqu'alors, toutes les machines étaient créées pour exécuter une seule et unique tâche, un seul calcul. En vulgarisant, une machine qui pouvait résoudre une addition ne pouvait pas résoudre une multiplication. Les machines étaient donc très limitées et leur coût était trop excessif pour pouvoir les utiliser couramment.

La machine de Turing est un concept abstrait qui a donné naissance à l'informatique que nous connaissons actuellement. En résumé, cette machine se compose d'un ruban **infini** de cases. Chaque case comporte un symbole que la machine connaît. La machine **lit les cases les unes après les autres et retient les cases dont elle a besoin** pour **effectuer le calcul décrit sur plusieurs cases**. Il s'agit de la première machine disposant d'une mémoire incorporée permettant de résoudre de multiples problèmes différents.

La machine de Turing n'a jamais été réellement implémentée, notamment à cause du ruban infini représentant la mémoire. Cependant, elle reste le **concept de base** des ordinateurs.

Nos ordinateurs actuels fonctionnent tous sur **l'architecture de Von Neumann**. John Von Neumann était un mathématicien et physicien né en 1903, ayant participé à la création des ordinateurs. Il proposa déjà à l'époque que les machines soient composées de quatre modules, comme le montre la figure suivante :

- **L'unité arithmétique et logique** qui effectue les opérations : ce sont nos processeurs aujourd'hui.
- **L'unité de contrôle** qui séquence les opérations, ce qui est aujourd'hui également intégré à nos processeurs.
- **La mémoire** pour stocker les programmes et les entrées, qui est devenue notre RAM et nos disques durs.
- **Les dispositifs d'entrée/sortie** pour que l'humain puisse donner les entrées à la machine, les claviers et les écrans principalement aujourd'hui.



Schématisation de l'arithmétique de Von Neumann

Cette architecture est bien celle sur laquelle se basent nos ordinateurs. Ce sont juste les composants électroniques, notamment les processeurs, qui deviennent de plus en plus puissants en termes de capacité de calcul.

Les processeurs ne sont pas les seuls à évoluer au cours du temps : la mémoire ne cesse d'augmenter en capacité de stockage, ce qui permet à nos programmes d'être de plus en plus élaborés. Mais quel est le lien entre programmes et mémoire de l'ordinateur ? En effet, le lien entre programmes et processeurs se devine aisément, mais celui avec la mémoire, pas forcément.

1.2 Implémentation de la mémoire

En tant qu'utilisateur, la mémoire représente la capacité du disque dur à stocker des fichiers sur la machine et donc à pouvoir s'en servir par la suite. Cette définition de la mémoire est juste mais également erronée. En effet, pour un informaticien, la mémoire n'est pas le disque dur.

1.2.1 Différentes mémoires

Un ordinateur possède plusieurs mémoires dont :

- **La mémoire vive ou RAM** : zone mémoire qui permet d'accéder à toutes les informations stockées et qui est volatile (qui s'efface au fur et à mesure et s'efface totalement lors de l'arrêt de la machine).
- **La mémoire morte ou ROM** : zone mémoire bas niveau qui stocke des informations qui ne peuvent pas changer, généralement pour retenir les instructions du fonctionnement des composants électroniques de l'ordinateur.
- **La mémoire cache** : zone mémoire qui sert à conserver des données souvent utilisées un court instant, elle fournit une grande rapidité pour accéder aux données mais elle est petite de taille car très coûteuse.

1.2.2 Programme et mémoire

Lorsque nous développons un programme, il est tout d'abord stocké sous forme d'un fichier sur le disque dur, tout comme un utilisateur stocke une photo sur son ordinateur pour la sauvegarder.

Mais lorsqu'un programme s'exécute sur un ordinateur, l'ordinateur doit l'analyser, retenir les instructions, ce que fait le programme, les résultats intermédiaires de ces instructions, les données sur lesquelles le programme travaille, etc.

Un programme ne prend donc pas que de la place sur le disque dur : il va **stocker au minima en mémoire vive** toutes les informations dont il a besoin pour s'exécuter. Or la capacité de la mémoire vive est largement plus petite que celle d'un disque dur.

C'est cette petite capacité de mémoire vive qui intéresse l'informaticien : les programmes que nous développons doivent optimiser l'organisation de cette zone mémoire. Et c'est principalement cette contrainte qui a permis aux langages de programmation d'évoluer pour devenir de plus en plus performants.

Nous développerons davantage cette partie dans le chapitre Passons en mode confirmé de cet ouvrage.

Mais avant de parler de programmation, commençons par parler de la manière de penser en développement informatique, la base de tout programme.

2. L'algorithmique, l'art de programmer

2.1 L'algorithmie, comment et pourquoi ?

Nous utilisons tous les jours des algorithmes sans même savoir que ce sont des algorithmes. Les **premiers algorithmes** formalisés remontent à l'**Antiquité** pour la gestion de l'eau dans les fontaines et depuis nous n'arrêtons pas d'en créer et de les appliquer.

2.1.1 Exemples de la vie courante

Indiquer à un ordinateur ce qu'il doit faire, donc lui donner un programme à exécuter, revient à donner à un humain la notice de montage d'un meuble en kit pour construire son étagère ou une recette pour pouvoir déguster une tarte aux pommes.

Nous avons tous connu ce moment de questionnement intense devant une notice de montage. Dans cette notice, nous retrouvons deux parties : l'ensemble des pièces fournies et une suite de schémas pour assembler les pièces. Ce document est en fait semblable à un algorithme et à un programme :

- La liste des pièces équivaut à l'ensemble des **variables** : 18 vis, 4 planches rectangulaires, un tournevis…
- Les schémas représentent les **instructions** : insérer les vis dans les trous des planches par exemple.

Regardons maintenant une deuxième analogie : la recette de cuisine. Vous aurez besoin d'une pâte brisée, d'un sachet de sucre vanillé, de 30 grammes de beurre et de 6 pommes. Nous pouvons voir la liste des ingrédients comme un ensemble de variables. Après la liste d'ingrédients viennent les étapes de la recette, donc les instructions :

- mettre la pâte brisée dans un moule à tarte ;
- éplucher les pommes ;
- couper les pommes en tranches ;
- placer les tranches sur la pâte ;

- faire fondre le beurre ;
- verser le beurre fondu sur la tarte ;
- verser le sucre vanillé sur la tarte ;
- préchauffer le four à 180° ;
- faire cuire la tarte 25 minutes au four.

Pour ceux qui n'ont jamais cuisiné ni monté un meuble en kit, un autre algorithme humain devrait vous parler : comment traverser à un passage piéton ? Lorsque nous traversons à un passage piéton, nous vérifions que le feu piéton est bien vert, qu'aucune voiture n'arrive sur le passage piéton, que le feu voiture est bien au rouge s'il y en a un, etc. Nous suivons donc des instructions que nous avons apprises pour assurer notre sécurité. Nous suivons donc un algorithme.

2.1.2 Algorithmes

Par définition, un algorithme est une **suite finie d'instructions** appliquant des **opérations non ambiguës** sur un **ensemble de variables**. Contrairement à la notice de montage et à la recette de cuisine, l'ordinateur comprend parfaitement un algorithme du premier coup et l'exécute sans jamais se tromper.

D'un point de vue plus général, un algorithme propose une **solution compréhensible et applicable** pour résoudre un problème, comme pour le passage piéton.

Voyez l'algorithmie comme le langage universel de tous les programmes et langages de programmation. Tout algorithme peut être traduit dans un programme et vice-versa.

Avec l'apprentissage de l'algorithmie, vous apprenez également comment communiquer avec l'ordinateur : ce qu'il peut comprendre, comment lui donner les informations, comment le guider dans l'exécution du programme.

Tout comme vous apprenez à un enfant à traverser une route, vous allez indiquer dans un algorithme ce que l'ordinateur peut faire ou ne pas faire, les conditions à tester, les comportements à répéter et dans quels cas les répéter, etc.

3. Les langages, la mise en œuvre

3.1 La programmation

L'ordinateur est un ensemble de composants électroniques qui, au fond, ne comprennent que deux choses : le courant passe ou le courant ne passe pas.

Ainsi le seul langage compris par les composants de la machine, que ce soient le processeur ou les mémoires, est le binaire. Uniquement deux valeurs sont possibles : 1 (le courant passe) et 0 (le courant ne passe pas).

Mais alors, comment communiquer de manière simple avec l'ordinateur ?

Au fur et à mesure du temps, l'informatique a implémenté de nombreuses surcouches au langage binaire pour faciliter la communication avec l'utilisateur : l'encodage du texte pour que l'être humain écrive dans un langage naturel, les systèmes d'exploitation comme Windows, Linux ou macOS...

Mais ces surcouches ne font qu'une traduction en binaire pour les composants électroniques. Tous vos fichiers numériques sont en fait une suite de 0 et de 1.

Il en va de même pour le développeur. Au début de l'informatique, le développement ressemblait plus à de l'électronique qu'à de l'informatique, puis sont apparues les cartes perforées pour stocker les programmes, puis les langages de programmation.

Les **langages de programmation** sont les langages de haut niveau qui permettent au développeur d'indiquer à l'ordinateur ce qu'il doit effectuer en **appliquant des algorithmes**.

Les langages de programmation ont une syntaxe proche du langage humain tout en intégrant les contraintes de la machine. Ils permettent de guider l'exécution d'un programme sans aucune ambiguïté et, donc, assurent le fonctionnement de nos instructions. Dans le cas où le programme ne fait pas ce qu'il devrait faire, le problème vient des instructions. Le problème est toujours entre le clavier et l'écran.

Remarque

Le premier programme a été inventé avant même que l'ordinateur soit créé. Ada Lovelace, comtesse anglaise née en 1815, développa le premier programme sur un ancêtre de l'ordinateur, la machine analytique de Charles Babbage. Il s'agit des premières instructions qu'un être humain ait données à une machine. Pour lui rendre hommage, le langage de programmation Ada porte son nom.

3.2 Les différents types de langages

La programmation est un domaine en perpétuelle évolution. Certains langages peuvent proposer des mises à jour plusieurs fois par an pour rendre vos programmes plus performants ou vous offrir plus de possibilités dans vos instructions.

Les premiers langages de programmation réels sont les langages procéduraux, ou impératifs, la base des autres styles de langages.

3.2.1 Programmation procédurale

La programmation procédurale consiste à écrire une suite d'instructions que l'ordinateur va exécuter les unes après les autres. Nous commencerons notre apprentissage grâce à ce style de programmation.

Les langages C, Python, Ada et Pascal sont des langages qui permettent de développer en procédural.

3.2.2 Programmation orientée objet

La programmation orientée objet se base sur la programmation procédurale en changeant l'organisation des instructions. Le programme n'est plus uniquement une suite d'instructions à exécuter, il est composé de plusieurs briques qui communiquent les unes avec les autres.

Nous retrouvons les langages C++, Java, Python ou C#, entre autres, pour programmer avec des objets.

Nous ferons une première introduction à ce paradigme de programmation lors du dernier chapitre de cet ouvrage.

3.2.3 Programmation fonctionnelle

La programmation fonctionnelle connaît un nouvel essor ces dernières années mais elle reste encore quelque peu exotique. Elle met la notion de fonction au centre du programme en bannissant les classes et les objets.

Les langages Scala ou Rust sont les deux exemples les plus utilisés en programmation fonctionnelle aujourd'hui.

3.3 Python

Nous avons fait le choix de mettre en pratique nos algorithmes avec le langage Python dans sa version 3. Ce choix a été guidé par les raisons suivantes :

- Il est gratuit.
- Sa syntaxe est simple.
- Il est multiplateforme : il peut s'exécuter sur les systèmes DOS (Windows) et Unix (Linux et macOS).
- Il est indépendant du système d'exploitation grâce à son interpréteur.
- Il évolue sans cesse grâce à une importante communauté très active.
- Il permet de programmer en procédurale et en orienté objet.

Nous n'en dirons pas plus sur ce langage dans cette introduction pour laisser le lecteur le découvrir au fur et à mesure des chapitres de cet ouvrage.

Chapitre 2
Les variables et opérateurs

1. Les variables simples

Reprenons les deux analogies du chapitre d'introduction. Avec la notice de montage du meuble en kit, nous pouvons dégager deux parties distinctes :

- La liste des pièces équivaut à l'ensemble des **variables** : 18 vis, 4 planches rectangulaires, 1 tournevis...
- Les schémas représentent les **instructions** : insérer les vis dans les trous des planches par exemple.

Le fait d'insérer les vis dans les planches peut être considéré comme un **opérateur** car il va modifier nos variables planches et vis en créant des planches avec des vis.

Regardons maintenant la deuxième analogie : la recette de cuisine. Vous aurez besoin d'1 pâte brisée, 1 sachet de sucre vanillé, de 30 grammes de beurre et de 6 pommes. Nous pouvons voir la liste des ingrédients comme un ensemble de variables qui auront toutes une **valeur** attribuée.

Nous pouvons traduire cette liste en algorithmie comme suit :

```
pate_brisee <-  1
sucre_vanille <-  1
beurre_en_grammes <-  30
pommes <-  6
```

Nous utilisons l'opérateur <- pour affecter une valeur à une variable : les pommes sont au nombre de six (`pommes <- 6`). Au fur et à mesure que nous allons intégrer les pommes à la tarte, le nombre de pommes restantes va diminuer. Une variable, comme son nom l'indique, a une valeur qui peut changer dans le temps et sert à mémoriser cette valeur avec un label donné.

Remarque

Par convention en algorithmie, les caractères accentués ne sont pas autorisés et le caractère "_" permet de séparer deux mots dans un nom de variable ou de programme pour qu'il soit plus lisible.

Le principe de variables permet à l'ordinateur de connaître et de stocker une valeur labellisée à tout instant lors de l'exécution du programme.

À la différence de la liste de courses et à la recette de cuisine, l'ordinateur a besoin en plus de son nom de connaître le type d'une variable, qui est appelé son **identifiant**.

L'identifiant d'une variable doit être **unique** au sein du programme ou de l'algorithme, sinon l'ordinateur ne pourra jamais savoir quelle variable manipuler. Imaginez-vous à une soirée où vous êtes confronté à trois personnes qui s'appellent Kévin. Si on vous demande d'aller chercher Kévin, lequel allez-vous choisir ? Sans information autre que des éléments de description tels que la couleur de cheveux, la taille, etc. vous ne pourrez jamais savoir quel est le Kévin dont on vous parle, cela va de même avec l'ordinateur.

1.1 Types, déclaration et affectation

L'ordinateur parle uniquement avec un langage lié aux mathématiques car le binaire est le cœur de son langage, tout est donc représenté par des octets. Heureusement, il existe des surcouches aux langages binaires pour nous rapprocher du langage naturel, nous ne développons plus en binaire depuis des décennies.

Parmi ces surcouches, il existe les types de données pour indiquer ce que la variable représente comme type d'information :

- les nombres ou numériques
- les caractères
- les booléens (le type le plus simple)
- les chaînes de caractères

En algorithmie, une variable a toujours le même type du début à la fin de l'algorithme. Le typage est dit **fort** de ce fait.

Explorons maintenant ces différents types de variables.

1.1.1 Les nombres ou numériques

En informatique, nous distinguons au moins deux types de numériques :

- les entiers ($\mathbb{N}$)
- les nombres décimaux, les réels (l'ensemble $\mathbb{R}$)

Ces deux types représentent des valeurs numériques positives, nulles ou négatives incluses entre $[-\infty; +\infty]$, en théorie pour l'algorithmie.

Les entiers sont des numériques sans chiffre après la virgule, ce que nous appelons communément les chiffres et nombres ronds. Un entier se déclare par son identifiant suivi du type `ENTIER` :

```
mon_entier : ENTIER
```

L'opérateur **:** permet à l'ordinateur de comprendre qu'après l'identifiant de la variable se trouve son type, ici `ENTIER`. Sans identifiant ou type, l'ordinateur ne peut pas comprendre que vous lui parlez d'une variable. Cette ligne, ou cette instruction plus précisément, est appelée la **déclaration de la variable**.

Les nombres à virgules ou réels se déclarent grâce au mot-clé `REEL` de la même manière que les nombres entiers (l'identifiant de la variable suivi de l'opérateur **:** et du type `REEL`) .

```
mon_reel : REEL
```

1.1.2 Les caractères

Le type CARACTERE correspond à un caractère unique. Sa valeur est toujours placée entre **simples quotes** (').

```
mon_carac : CARACTERE <-  'a'
mon_chiffre : CARACTERE <-  '7'
```

Dans l'algorithme précédent, la variable mon_carac représente le caractère "a" et mon_chiffre le caractère "7". Nous remarquons également que nous pouvons déclarer une variable et lui donner une valeur dans la même instruction. Il est aussi possible d'écrire cela en deux lignes afin d'afficher la valeur plus loin dans le code. Nous rappelons que l'affectation se traduit par l'opérateur <-.

Un caractère peut représenter n'importe quelle touche du clavier. Cependant, si le caractère représente un chiffre, il ne sera pas possible de faire des calculs sur ce dernier.

Remarque

Un caractère est forcément représenté par un nombre binaire. Pour ce faire, une convention existe : la table ASCII pour les caractères anglophones et la table ASCII étendue qui lui ajoute les caractères accentués et les caractères spéciaux à une langue. Ces deux tables indiquent quelle est la valeur binaire pour chaque caractère utilisable en informatique comme le montre la figure ci-après.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[END OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Extrait de la table ASCII

1.1.3 Les booléens

Les booléens sont le type le plus important en informatique et pourtant le plus simple.

Lorsque vous voulez traverser à un passage piéton avec un feu de circulation, vous avez deux voyants, un rouge et un vert, qui vous indiquent si vous pouvez vous engager ou non. Ces deux voyants sont des booléens : soit vous pouvez traverser, soit non, soit ils sont allumés, soit ils sont éteints.

Un booléen ne peut prendre que deux valeurs : `VRAI` ou `FAUX`. Ce type permet à l'ordinateur de savoir s'il peut exécuter telle instruction, s'il a fini une tâche, etc.

Voici un algorithme qui déclare et initialise deux booléens grâce au mot-clé `BOOLEEN`.

```
mon_vrai : BOOLEEN <-  VRAI
mon_faux : BOOLEEN <-  FAUX
```

1.1.4 Les chaînes de caractères

Nous ne sommes plus dans les années 1980 depuis longtemps maintenant, l'ordinateur est manipulé par des utilisateurs non plus experts de l'informatique et doit pouvoir se faire comprendre de l'être humain facilement. C'est l'objectif du type `CHAINE` qui représente du texte, appelé en informatique chaîne de caractères (plusieurs caractères qui se suivent).

La valeur d'une chaîne est toujours placée entre doubles quotes :

```
texte : CHAINE <- "les bases avant et vous ferez votre premier
programme après !"
```

Les manipulations possibles sur ce type seront vues dans une section dédiée de ce chapitre un peu plus tard. Pour le moment, nous allons utiliser les chaînes de caractères uniquement comme des indications pour l'utilisateur.

Nous sommes maintenant prêts pour écrire notre premier algorithme !

1.2 Saisie et affichage

Pour pouvoir interagir avec un utilisateur, nous devons lui demander les valeurs sur lesquelles notre algorithme doit travailler et lui indiquer les changements de valeurs afin de lui transmettre les informations qu'il demande. La saisie sert à récupérer les valeurs grâce à une entrée clavier (l'utilisateur rentre sur le clavier la valeur demandée) et l'affichage envoie sur une sortie, l'écran, une valeur sous forme de texte pour l'utilisateur.

Avant de communiquer avec l'utilisateur, il nous faut voir comment définir un algorithme. La machine est stupide, elle ne fait que ce que nous lui disons ! De ce fait, nous devons lui indiquer quand commence l'algorithme (`DEBUT`) et quand il finit (`FIN`).

Tout comme les variables, chaque algorithme doit avoir un identifiant unique pour que l'ordinateur sache le trouver pour le faire tourner et il doit être de type `PROGRAMME`.

Nous devons également lui définir clairement où sont les variables : les variables sont toujours déclarées en début d'algorithme dans la section `VAR`. En informatique, ce type de section est appelé **bloc**. Vous ne pouvez pas utiliser une variable sans la déclarer dans ce bloc.

Voici notre premier algorithme qui déclare une variable x et lui affecte la valeur 3,67 :

```
PROGRAMME Exemple
VAR
   x : REEL
DEBUT
   x <- 3,67
FIN
```

Remarque

Les variables doivent être déclarées obligatoirement dans le bloc `VAR` mais leur affectation initiale peut se faire lors de la déclaration ou dans l'algorithme.

Remarque

Une variable qui n'a jamais été initialisée avec une première valeur ne peut pas être utilisée dans des instructions autre que son initialisation (première affectation de valeur).

Revenons à l'objectif principal de cette section : échanger des informations avec l'utilisateur. Il existe l'opérateur `ECRIRE()` pour afficher une valeur et l'opérateur `LIRE()` pour récupérer une valeur. N'oubliez pas que vous décrivez le comportement de l'ordinateur et non celui de l'utilisateur. Faisons saisir à notre utilisateur un nombre entier :

```
PROGRAMME Afficher_saisie
VAR
   nombre : ENTIER
DEBUT
   ECRIRE("Entrez un nombre entier")
   nombre <- LIRE()
   ECRIRE("Votre nombre est ")
   ECRIRE(nombre)
FIN
```

Remarque

En algorithmie, les opérations et opérateurs "complexes" sont toujours représentés par des verbes à l'infinitif.

L'opérateur `ECRIRE` peut prendre un ou plusieurs **paramètres**, c'est-à-dire entre les parenthèses qui le suivent, soit une valeur, soit une variable, soit plusieurs valeurs et/ou variables. Ce sont ces paramètres qui seront affichés par l'utilisateur sur l'écran, que nous appelons la **console** dans le jargon informatique. Dans tous les cas, cet opérateur ne peut afficher qu'une valeur, qu'elle soit affectée ou non à une variable. Voyez les paramètres comme la nourriture que nous donnons à manger à l'opérateur.

L'opérateur `ECRIRE` peut regrouper plusieurs valeurs en les affichant sur une ligne. Pour ce faire, vous devez les lister en les séparant par des virgules. Pour afficher une phrase compréhensible à l'utilisateur, vous pouvez utiliser cet opérateur comme suit : `ECRIRE("Le nombre est " , nombre)`.

Remarque

L'espace qui suit une parenthèse est facultatif, nous l'ajoutons uniquement pour la lecture humaine que ce soit dans un algorithme ou dans un programme.

L'opérateur `LIRE` sert à affecter une variable. Il ne prend aucun paramètre (parenthèses toujours vides juste après).

Pour que l'ordinateur comprenne la **fin d'une instruction**, ici une déclaration, deux affichages, une affection puis un affichage, nous **allons à la ligne** à la fin de chaque instruction.

Il est également important de respecter le principe **d'indentation** : dans chaque bloc, ici `VAR` et `DEBUT`, nous commençons nos lignes par une tabulation. Cela permet de rendre l'algorithme plus lisible pour l'œil humain, donc de le rendre plus maintenable pour le faire évoluer au cours du temps, tout comme un programme.

1.3 Les constantes

Terminons cette présentation des variables par les constantes. Il est quelquefois intéressant de bloquer la valeur d'une variable pour garantir l'exécution de l'algorithme et donc du programme qui va en découler. C'est le rôle des constantes.

Une constante est une variable dont la valeur ne peut pas être modifiée après son initialisation. Son affection est obligatoirement faite pendant sa déclaration dans un bloc `CONST`. Hormis ces deux contraintes, elle s'utilise par la suite comme une variable :

```
PROGRAMME Afficher_pi
CONST
   pi <- 3,14 : REEL
VAR
   n : ENTIER
DEBUT
   ECRIRE(pi)
   ...
FIN
```

Une constante ne peut pas :

- recevoir la valeur de l'opérateur `LIRE()` ;
- être affectée après son initialisation avec l'opérateur `<-`.

2. Utiliser les variables

Les variables ne servent à rien si nous ne pouvons pas les manipuler. Pour ce faire, nous allons utiliser des opérateurs qui sont relatifs au type de la variable.

Remarque

Tous les opérateurs fonctionnent soit avec des variables, soit avec des valeurs, soit avec une variable et une valeur.

Un opérateur est dit binaire s'il a deux opérandes, c'est-à-dire qu'il fonctionne avec deux valeurs. Un opérateur unaire n'a qu'un opérande.

2.1 Opérateurs mathématiques communs

Les opérateurs mathématiques communs sont utilisables sur les types `ENTIER` et `REEL`. Ils représentent les calculs mathématiques basiques :

- Addition avec le symbole **+**.
- Soustraction avec le symbole **-**.
- Multiplication avec le symbole **x**.

```
PROGRAMME Exemple_operateur_math
VAR
   a <- 17 : ENTIER
   b <- 2 : ENTIER
   z <- 1.1 : REEL
   y <- 2.87 : REEL
   res : ENTIER
DEBUT
   res <- a + b
   ECRIRE("L'addition de a et b vaut ",res)
   res <- a - b
   ECRIRE("La soustraction de a et b vaut ", res
```

```
    res <- a x b
    ECRIRE("La multiplication de a et b vaut ", res)
    ECRIRE(z + y)
    ECRIRE(z - y)
    ECRIRE(z x 3.89)
FIN
```

Remarque

Vous pouvez remarquer dans l'algorithme précédent que l'opérateur `ECRIRE` permet d'afficher la valeur du résultat d'une opération sans l'obligation de la stocker dans une autre variable.

2.2 Opérateurs spécifiques aux entiers

Les entiers ont deux opérateurs dédiés :

- La division avec le symbole `DIV`.
- Le reste de la division euclidienne avec le symbole `MOD` pour modulo.

```
PROGRAMME Operateurs_entie
VAR
    n : ENTIER <- 30
    m : ENTIER <- 4
DEBUT
    ECRIRE("n divise par m vaut")
    ECRIRE(n DIV m)          // affiche 7
    ECRIRE("Le reste de la division de n par m vaut ")
    ECRIRE(n MOD m)          // affiche 2
FIN
```

Remarque

Pour ajouter des commentaires dans notre algorithme afin d'indiquer des informations au lecteur, nous utilisons le symbole `//`. Grâce à ce symbole, un autre développeur comprend qu'il ne s'agit pas d'une instruction mais d'une indication sur le comportement de l'algorithme.

2.3 Opérateur spécifique aux réels

Comme les entiers ont une division entière, les réels ont une division réelle avec l'opérateur / classique.

```
PROGRAMME Operateur_reel
VAR
   n : REEL <- 30.87
   m : REEL <- 4.65
DEBUT
   ECRIRE("n divisé par m vaut")
   ECRIRE(n / m)          // affiche 6,6387096774
FIN
```

2.4 Opérateurs de comparaison

Les opérateurs de comparaison permettent de comparer deux valeurs numériques :

- Égalité avec le symbole =.
- Différence avec le symbole ≠.
- Plus grand que avec le symbole >.
- Plus petit que avec le symbole <.
- Plus grand ou égal à avec le symbole >=.
- Plus petit ou égal à avec le symbole <=.

Ces opérateurs binaires retournent toujours un booléen, ce qui est tout à fait logique, le résultat d'une comparaison étant toujours oui ou non, donc VRAI ou FAUX.

```
PROGRAMME Exemple_operateurs_comparaison
VAR
   a : ENTIER <- 1
   b : ENTIER <- 2

DEBUT
   ECRIRE("Egalité : ", a = b) // FAUX
   ECRIRE("Différence : ", a ? b) // VRAI
   ECRIRE("Infériorité : ", a < b) // VRAI
```

```
    ECRIRE("Supériorité : ", a > b) // FAUX
    ECRIRE("Infériorité ou égalité : ", a <= b) // VRAI
    ECRIRE("Supériorité ou égalité : ", a >= b) // FAUX
FIN
```

Remarque

En algorithmie, nous ne pouvons comparer que deux valeurs de même type.

2.5 Opérateurs logiques pour les booléens

Retournons un peu à la vie réelle pour expliquer les opérateurs sur les booléens. Lorsque vous décidez de sortir de chez vous par exemple, vous prenez votre décision en fonction de plusieurs critères. Vous regardez si la sortie vous intéresse ou si elle est nécessaire et si le temps vous permet de sortir. Vous effectuez donc des calculs logiques en prenant en compte ces critères. Ces liens entre les conditions sont représentés en informatique par les opérateurs logiques.

L'opérateur binaire `OU` renvoie `VRAI` si au moins un des deux opérandes est `VRAI`.

L'opérateur binaire `ET` renvoie `VRAI` si les deux opérandes sont `VRAI`.

L'opérateur unaire `NON` renvoie la valeur inverse du booléen : `VRAI` devient `FAUX` et vice versa.

```
PROGRAMME Exemple_operateurs_logiques
VAR
   interet : BOOLEEN <- FAUX
   necessaire : BOOLEN <- VRAI
DEBUT
   ECRIRE("OU logique : ", interet OU necessaire) // VRAI
   ECRIRE("ET logique : ", interet ET necessaire) // FAUX
   ECRIRE("NON logique : ", NON necessaire) // FAUX
FIN
```

La logique de ces trois opérateurs sera un peu plus approfondie dans le chapitre suivant.

2.6 Opérateurs sur les caractères

Comme nous l'avons remarqué précédemment, les caractères sont représentés dans un ordre donné par un code caractère. De ce fait, nous pouvons utiliser les opérateurs de comparaison vus précédemment avec les caractères, comme avec les types entiers et réels :

- Égalité avec le symbole =
- Différence avec le symbole ≠.
- Plus grand que avec le symbole >.
- Plus petit que avec le symbole <.
- Plus grand ou égal à avec le symbole >=.
- Plus petit ou égal à avec le symbole <=.

Le plus petit caractère est celui qui arrive le premier dans la table et donc le plus grand celui qui arrive en dernier. Notons que les majuscules sont toujours plus petites que les minuscules et qu'il existe bien une différence entre 'a' et 'A', ce ne sont pas les mêmes caractères.

```
PROGRAMME Exemple_comparaison_caractere
VAR
   a : CARACTERE <- 'a'
   b : CARACTERE <- 'b'
   carac : CARACTERE <- 'A'
DEBUT
   ECRIRE("Egalité : ", a = carac) // FAUX
   ECRIRE("Différence : ", a ? carac) // VRAI
   ECRIRE("Infériorité : ", a < b) // VRAI
   ECRIRE(("Supériorité : ", a > b) // FAUX
   ECRIRE("Infériorité ou égalité : ", a <= b) // VRAI
   ECRIRE("Supériorité ou égalité : ", a >= b) // FAUX
FIN
```

3. Les opérations sur les chaînes

Les chaînes de caractères sont des types complexes. En effet, contrairement aux autres types que nous avons vus, elles sont constituées de plusieurs valeurs (plusieurs caractères) les unes après les autres et non pas d'une seule.

De ce fait, nous ne pouvons pas les manipuler aussi simplement que les autres types avec de simples opérateurs.

En algorithmie, nous possédons pour traiter les chaînes de caractères :

- La **concaténation** : ajouter une chaîne à la suite d'une autre.
- L'**extraction** : créer une chaîne à partir d'une autre.
- La **longueur** : donner le nombre de caractères d'une chaîne.

3.1 Concaténation

Pour coller deux chaînes de caractères ensemble, nous utilisons l'opérateur de concaténation CONCATENER. Cet opérateur prend deux chaînes de caractères en paramètres pour produire une nouvelle chaîne qui aura comme valeur la première chaîne suivie de la deuxième :

```
PROGRAMME Exemple_Concat
VAR
   chaine1 : CHAINE <- "Hello"
   chaine2 : CHAINE <- "World"
   res : CHAINE
DEBUT
   res <- CONCATENER(chaine1, chaine2)
   ECRIRE(res)    // HelloWord
   res <- CONCATENER(chaine1, " ")
   ECRIRE(res)    // Hello
    res <- CONCATENER(res, chaine2)
    ECRIRE(res)    // Hello Word
FIN
```

Comme le montre l'algorithme précédent, la gestion des espaces est à la charge du développeur : l'algorithme ne comprend pas quand, ou si, un espace est requis.

3.2 Extraction

L'opérateur d'extraction EXTRAIRE permet de créer une nouvelle chaîne de caractères à partir d'une autre. Nous lui indiquons à partir de quel caractère nous voulons commencer et combien de caractères nous voulons recopier. Cet opérateur extrait donc une sous-chaîne d'une autre chaîne donnée.

EXTRAIRE prend trois paramètres :

- La **chaîne existante**.
- La **position du caractère** à partir duquel nous voulons commencer à recopier la chaîne.
- Le **nombre de caractères** à recopier dans la sous-chaîne à partir de la position indiquée.

```
PROGRAMME Exemple_Extraire
VAR
   chaine1 : CHAINE <- "Hello World"
   res : CHAINE
DEBUT
   res <- EXTRAIRE(chaine1, 7, 2)
   ECRIRE(res)    // Wor
FIN
```

Notons qu'en algorithmie, le premier caractère d'une chaîne est à la position 1. En informatique nous appelons cette position **l'indice** du caractère.

3.3 Longueur

Le dernier opérateur permet de récupérer le nombre de caractères d'une chaîne de caractères. Il s'agit de LONGUEUR qui retourne un entier et prend en paramètre la chaîne dont nous voulons avoir le nombre de caractères.

```
PROGRAMME Exemple_Extraire
VAR
   chaine1 : CHAINE <- "Hello World"
   res : ENTIER
DEBUT
   res <- LONGUEUR(chaine1)
   ECRIRE(res)    // 11
FIN
```

Cet opérateur ne vous semble pas forcément utile maintenant mais nous allons voir au cours de cet ouvrage qu'il est primordial dans tout algorithme ou programme qui manipulent des chaînes de caractères.

4. Et avec les langages ?

L'algorithmie se veut représenter le plus de langages possibles. Cependant, nous allons confirmer dans cette section la fameuse expression "il existe toujours un fossé entre la théorie et la pratique". En effet, l'algorithmie est destinée à être une base commune pour la majorité des langages de programmation mais chacun possède ses propres mécanismes de gestion internes qui peuvent grandement modifier le code par rapport à l'algorithme. Nous verrons dans cette section quelques-uns de ces fossés avant de commencer l'apprentissage du langage Python.

4.1 Typage

Actuellement, en programmation, nous pouvons distinguer quatre classes intéressantes de langages :

- Les langages à **typage statique**.
- Les langages à **typage dynamique**.
- Les langages **fortement typé**.
- Les langages **faiblement typés**.

Les langages peuvent imposer de déclarer ou non le type des variables d'un programme (typage statique ou dynamique). Ils peuvent également permettre ou non de changer le type d'une variable au cours de l'exécution d'un programme (faiblement ou fortement typé).

4.1.1 Les langages à typage statique

Le typage statique indique que le langage impose pour chaque variable la déclaration de son type lors de la déclaration de la variable, tout comme en algorithmie. Son initialisation peut intervenir n'importe quand après sa déclaration.

Nous retrouvons dans ce type de langage des langages tels que C, C++, Java, C# et TypeScript par exemple.

Remarquons que le langage C est l'un des rares langages qui imposent de déclarer toutes les variables en début de programme comme en algorithmie.

4.1.2 Les langages à typage dynamique

Dans un langage à typage dynamique, le langage détermine le type de la variable grâce à son initialisation. Si nous affectons 3 à la variable n, le langage comprendra automatiquement que cette variable est de type entier. Il s'agit de la première grosse différence entre la pratique et la théorie.

Ces types de langages respectent uniquement le principe de l'algorithmie qu'une variable doit être initialisée avant d'être utilisée. Ils forcent même ce principe car la déclaration d'une variable impose forcément son initialisation !

Parmi les langages à typage dynamique, nous pouvons retrouver le Python, le PHP et le JavaScript par exemple.

4.1.3 Langages fortement typés

Les langages fortement typés imposent que, lorsqu'une variable est déclarée avec un type, cette variable gardera le même type tout au long de l'exécution du programme. Cela doit vous sembler tout à fait logique avec la lecture de ce chapitre : un entier reste un entier, un réel reste un réel, etc.

Le PHP, le C, le C++, le Java et le JavaScript font partie de ce type de langage.

4.1.4 Langages faiblement typés

Encore une énorme différence entre la théorie et la pratique, les langages faiblement typés permettent à une variable de changer de type au cours de l'exécution du programme. Ce sont généralement des langages à typage dynamique.

Prenons un exemple. Une variable a prend comme valeur 2 en début de programme. Le développeur peut modifier sa valeur avec la valeur "hello", changeant ainsi le type de a d'entier à chaîne de caractères.

Vous pouvez retrouver le Python, ou le JavaScript par exemple, comme langages faiblement typés.

Remarque

Ces quatre classes de langage peuvent vous paraître compliquées à gérer au départ mais rassurez-vous, une fois que vous commencerez à programmer, le type de typage deviendra naturel et vous ne vous poserez plus de questions.

4.2 Opérateurs

La plupart des langages actuels utilisent les opérateurs classiques d'une calculatrice pour les calculs mathématiques : +, - , *, /. Pour la division, vous pouvez retrouver la division entière avec le symbole // et la division réelle avec le symbole /. Le reste de la division euclidienne, le modulo, est généralement représenté par l'opérateur %. Vous retrouvez également une affectation plus classique avec le symbole = au lieu de la flèche.

Vous devez donc vous demander maintenant comment gérer l'égalité. Tout simplement avec le double égal : ==. Et la différence ? Avec !=. Les opérateurs d'infériorité et de supériorité restent les mêmes qu'en algorithmie.

Remarque

En programmation, le ! représente le NON logique. La différence étant l'inverse de l'égalité, donc un non égal, il nous suffit de faire suivre un ! par un = pour la représenter.

Quant aux opérateurs logiques, le ET est souvent représenté par && ou AND, le OU par | | ou OR et le NON par ! ou NOT.

Les opérateurs sur les chaînes de caractères peuvent être cependant très différents d'un langage à un autre. En effet, les chaînes sont des types complexes qui ne sont pas implémentées avec la même logique dans tous les langages ce qui implique des opérateurs très différents. Cependant vous retrouvez toujours les trois opérateurs de base CONCATENER, EXTRAIRE et LONGUEUR.

Vous verrez par la suite de ce chapitre que la traduction entre algorithme et programme n'est pas si compliquée, la gymnastique requise devient très rapidement un réflexe.

4.3 Gestion de la mémoire

Nous proposons dans cette partie une rapide introduction à la gestion de la mémoire, nous rentrerons dans les détails avec le chapitre Passons en mode confirmé dans la section Les pointeurs et références.

L'un des soucis majeurs de la programmation est l'utilisation et la gestion de la mémoire. La mémoire d'un ordinateur n'étant pas infinie, à la différence de la machine de Turing, il ne peut pas tout retenir.

Nous pouvons comparer la mémoire de la machine à une commode avec plusieurs tiroirs. Nous rangeons nos affaires dans ces tiroirs comme le programme stocke les instructions et les valeurs des variables. Que se passe-t-il quand les tiroirs sont tous pleins ? Nous ne pouvons plus y ranger d'affaires, tout comme le programme ne peut plus rien stocker dans sa mémoire.

Pour vulgariser les aspects techniques, chaque variable et chaque instruction prend de la place en mémoire. La question est : quand l'ordinateur peut-il effacer une variable ou instruction en mémoire ?

La réponse se trouve dans le langage utilisé pour programmer. Chaque langage a sa propre gestion, ses propres méthodes pour nettoyer la mémoire et ainsi faire que la mémoire de l'ordinateur n'explose pas sous le poids des données. Certains langages comme le C ou le C++ demandent au développeur de leur indiquer quand la mémoire doit être vidée et d'autres le font automatiquement (avec le Garbage Collector de Java par exemple).

4.4 La gestion des réels

En algorithmie, nous n'avons que le type `REEL` pour représenter les nombres à virgules. En programmation, une question peut se poser selon le langage : flottant ou décimal ?

Ces deux types représentent le type `REEL` en programmation. Il y a cependant une légère nuance entre les deux, nuance qui peut provoquer quelques surprises et sueurs froides au développeur.

Le **flottant** est un nombre réel qui est représenté en machine avec une **écriture scientifique binaire**. Sans rentrer dans les détails, le flottant est calculé pour ne garder que les puissances significatives du réel. Ce calcul rend donc le flottant quelques fois inexact par rapport à la valeur attendu. Par exemple, nous voulons stocker 0,1 avec un flottant et en l'affichant, nous obtenons 0,100000000001, ce qui peut donc provoquer quelques erreurs si nous n'y prêtons pas gare. Cette erreur de précision vient du calcul que fait la machine pour stocker notre valeur réelle.

Le **décimal** est un nombre à **virgule fixe**. Ici la machine stocke aussi les chiffres avant la virgule et ceux après tels quels. Le décimal apporte donc une valeur exacte contrairement au flottant. Cependant, il est largement plus gourmand en mémoire.

5. Python et les types

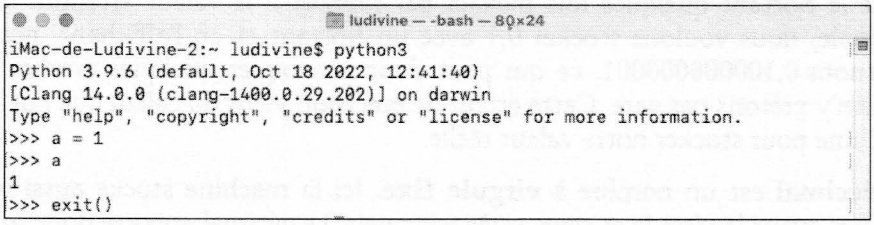
Maintenant que vous avez toutes les clés de la logique de programmation, entrons dans le vif du sujet en commençant par installer Python pour pouvoir écrire les programmes (ou scripts).

5.1 Installation

Python n'est installé d'office que sur les ordinateurs possédant comme système d'exploitation Linux ou macOS.

Pour les développeurs qui sont sous Windows, vous devez installer Python en suivant les instructions sur le site officiel : https://www.python.org/

- Pour vérifier que Python est bien installé, lancez une console/un terminal ou cmd et tapez la ligne de commande `python` (ou `python3` selon l'installation). Vous devriez avoir accès à ce que nous appelons la calculette Python, comme le montre la figure suivante. Cette calculette permet de lancer rapidement des instructions en Python, notamment pour les tester rapidement. Pour sortir de la calculette, tapez l'instruction `exit()`.

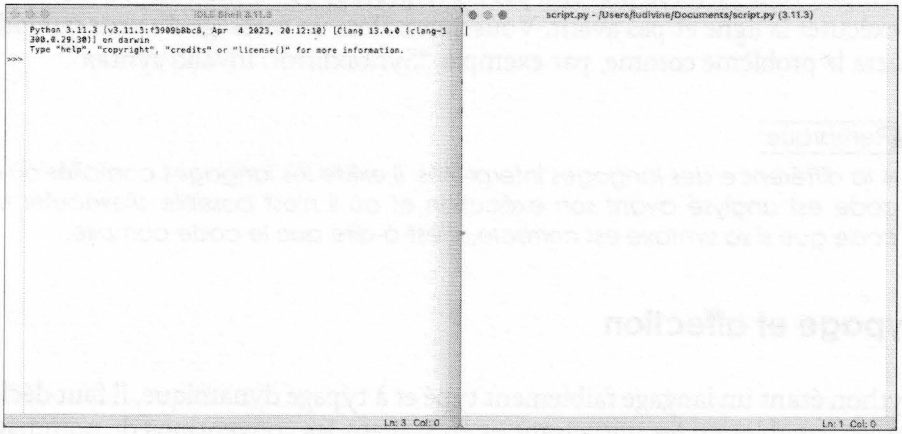


Calculette Python

En installant Python, vous installez automatiquement IDLE, l'environnement de développement intégré (IDE), pour Python. Cet IDE est composé de deux fenêtres, comme le montre la figure ci-dessous : la calculette Python qui va exécuter votre script (la fenêtre de gauche) et votre script (la fenêtre de droite). Ainsi, vous pouvez voir en même temps votre code d'un côté et son résultat de l'autre sur votre écran.

Remarque

Un IDE est logiciel qui va vous aider et vous conseiller pour écrire votre programmation. Il facilite votre travail en connaissant la syntaxe du langage dans lequel vous programmez. Ainsi, il indiquera vos erreurs plus rapidement et il vous permettra de lire plus rapidement votre code grâce à la coloration syntaxique. Certains IDE vous proposent en plus de compléter automatiquement votre code avec un système d'autocomplétion pseudo-intelligente.



IDLE

- Pour créer un nouveau script, allez dans le menu **Fichier - Nouveau** et enregistrez votre script avec l'extension .py.
- Cliquez sur le menu **Run** pour exécuter votre code.

Vous n'êtes pas dans l'obligation d'utiliser IDLE. Vous pouvez également coder avec SublimeText ou PyCharm par exemple. Il existe un assez grand nombre d'IDE pour Python, à vous de choisir celui qui vous convient le mieux.

Remarque

Il est conseillé d'installer l'avant-dernière version du langage afin d'assurer la compatibilité avec les librairies tierces pour les jours futurs où votre niveau vous permettra de les utiliser pour aller plus loin.

5.2 Langage interprété

Python est un langage faiblement typé et à typage dynamique, comme nous l'avons expliqué précédemment.

De plus, Python est un langage interprété. Cela veut dire que lorsque vous exécutez un script, les **instructions sont lues et analysées à leur lecture** et uniquement à ce moment. Ainsi, si vous avez une erreur de syntaxe par exemple, vous ne serez prévenu par l'interpréteur qu'au moment où il essaiera d'exécuter la ligne et pas avant. Vous aurez alors un message d'erreur qui indiquera le problème comme, par exemple, "SyntaxError: invalid syntax".

Remarque

À la différence des langages interprétés, il existe les langages compilés où le code est analysé avant son exécution et où il n'est possible d'exécuter un code que si sa syntaxe est correcte, c'est-à-dire que le code compile.

5.3 Typage et affection

Python étant un langage faiblement typé et à typage dynamique, il faut déclarer les variables en les initialisant et en suivant les conventions de nommage Python.

Remarque

*Python est un langage dit **sensible à la casse**, c'est-à-dire que l'identifiant a n'est pas le même que l'identifiant A, les majuscules et minuscules ont donc leur importance.*

5.3.1 Convention de nommage

Pour être un bon développeur Python, il est demandé de suivre la convention suivante :

- Identifiant : suite non vide de caractères, de longueur quelconque, formée d'un caractère de début et de 0 ou plusieurs caractères de continuation : id est correct contrairement à 1id.
- Ne pas utiliser de caractères accentués : numerique est correct, numérique ne l'est pas.
- MAJUSCULES ou UPPER_CASE pour les constantes : MA_CONSTANTE au lieu de maconstante.
- TitleCase pour les classes : MaClasse et non maClasse.
- camelCase pour les identifiants des fonctions, méthodes et interfaces graphiques : maFonction est correct contrairement à Mafonction ou ma-fonction.
- unmodule pour les modules : monmod et non mon-module ou MonModule.
- minuscule ou snake_case pour tous les autres identifiants : ma-variable ou mon-entier sont corrects et pas ma_variable ou encore maVariable.

Ne prenez pas peur sur les termes techniques que nous n'avons pas encore abordés, nous les verrons dans la suite de cet ouvrage.

La seule condition de syntaxe que vous êtes obligé de suivre est la suivante : un identifiant commence toujours par une lettre suivie ou non d'une suite de caractères hors l'espace, le point, le # et @.

5.3.2 Affection et commentaire

Comme Python est typé dynamiquement, vous devez initialiser vos variables pour les déclarer avec l'opérateur =. Les commentaires sont devancés par un #.

Les variables peuvent être déclarées à n'importe quelle ligne du script du moment qu'elles sont déclarées avant d'être utilisées.

Comme en algorithmie, la fin d'une instruction est le saut de ligne.

Python accepte les types suivants :

- **int** : nombres entiers.
- **float** : nombres décimaux.
- **complex** : nombres complexes (un réel et un imaginaire).
- **bool** : booléens (deux valeurs possibles `True` et `False`).
- **str** : chaînes de caractères (les caractères sont une chaîne d'un caractère).

```
entier = 1
mon_reel = 3.4
PI = 3.14 # fausse constante, juste une convention
hello = "hello world"
coucou = 'coucou'
b = True
nop = False
a, b = 1, 2 # a vaut 1 et b vaut 2
```

Nous remarquons que les chaînes peuvent être **entre simples ou doubles quotes**, peu importe.

Il est également possible en Python de faire des déclarations multiples en une simple instruction grâce à la virgule entre les variables et celle entre les valeurs. Les valeurs seront affectées dans les variables dans l'ordre de déclaration.

Les constantes en Python n'existent pas réellement. Nous indiquons que la variable est une constante en mettant son identifiant en majuscule, mais le langage ne nous empêche pas par la suite d'en modifier sa valeur.

5.3.3 Opérateurs sur les types

Python ne nous forçant pas à indiquer le type de la variable, il possède des opérateurs ou plutôt des fonctions pour manipuler les types de variables :

- `type(a)` : retourne le type de `a`.
- `eval(a)` : retourne la valeur de `a` interprété par l'interpréteur.
- Conversion de `a` :
  - `int(a)` pour transformer `a` en entier.
  - `float(a)` pour transformer `a` en réel.
  - `str(a)` pour transformer `a` en chaîne de caractères.

- `bool(a)` pour transformer `a` en booléen.
- `complex(r,i)` pour transformer `r` et `i` en complexe.

```
s = "7"
entier = eval(s) # entier vaut 7
entier = int('1') # entier vaut1
```

5.4 Un premier script

Maintenant que nous savons déclarer des variables, il nous manque la traduction de `LIRE` et `ECRIRE` en Python pour écrire notre premier script.

5.4.1 ECRIRE en Python

La fonction `print` affiche à l'écran les expressions passées en paramètres avec la syntaxe suivante :

```
print(expr_1,expr_2,...,sep=' ',end='\n')
```

- Les expressions en paramètres peuvent être des valeurs ou des variables, peu importe, sachant que seules les valeurs seront affichées et non le nom des variables.
- `sep` (optionnel) indique ce qui va séparer ces expressions, par défaut il s'agit d'un espace.
- `end` (optionnel) indique ce qui sera affiché après toutes les expressions, par défaut il s'agit d'un **retour à ligne** (`\n`).

```
a = 1
x = 8.7
s = "hello"
print("mes variables sont a avec la valeur", a, 'b
avec la valeur',b,'et s =',s)
# affiche "mes variables sont a avec la valeur 1 b
avec la valeur 8.7 et s hello"
print("a sans espace par defaut", a, sep='')
# affiche "a sans espace par defaut1"
print("sans saut ", end="")
print("de ligne")
# affiche sous "saut de ligne"
```

5.4.2 LIRE en Python

La fonction `input` permet de lire l'entrée clavier. Elle est donc la traduction de `LIRE` de l'algorithmie.

Par défaut, `input` retourne une chaîne de caractères de type `str` voulant dire string (chaîne en français) :

```
var = input(expression)
```

La fonction `input` retourne une valeur de type `str`, pensez donc à la transformer en un autre type au besoin.

L'`expression` est facultative, elle représente la demande de l'entrée clavier.

```
entree = input('Veuillez entrer un entier') # entree est une chaîne
print(type(entree))   # <class str>
entree = int(entree) # entree est un entier, eval(entree)
pour être plus générique
print(type(entree))   # <class 'int'>
```

5.5 Opérateurs

5.5.1 Opérateurs arithmétiques

Pour les variables de types numériques, vous pouvez utiliser les opérateurs suivants :

- Addition de x et y : **x+y**
- Soustraction de y à x : **x-y**
- Multiplication de x et y : **x*y**
- Élévation de x à la puissance y : **x**y**.
- Quotient (division) réel de x par y : **x/y**.
- Quotient (division) entier de x par y : **x//y**.
- Reste du quotient entier (modulo) de x par y : **x%y**.

Il existe également en Python des opérateurs d'affectation qui effectuent un calcul en même temps que cette affection :

- **x**+=**y** équivaut à x = x + y
- **x**-=**y** équivaut à x = x - y
- **x***=**y** équivaut à x = x * y
- **x**/=**y** équivaut à x = x / y
- **x**//=**y** équivaut à x = x // y
- **x**%=**y** équivaut à x = x % y

```
a = 1
b = 2
c = a + b
print("c vaut", c)    # affiche "c vaut 3"
a += b
print("a vaut", a)    # affiche "a vaut 3"
```

Remarque

L'opérateur + s'applique aussi sur les booléens et effectue un OU logique.

5.5.2 Opérateurs de comparaison

Hormis les opérateurs d'égalité et de différence, les opérateurs de comparaison sont les mêmes que ceux des algorithmes : <, >, <=, >=.

L'égalité est symbolisée par le double égal : ==.

La différence est symbolisée par le non égal : !=.

```
print(1 == 1)    # affiche True
print(1 != 1)    # affiche False
print(1 > 1)     # affiche False
```

5.5.3 Opérateurs logiques

Les opérateurs logiques en Python sont simplement la traduction en anglais des opérateurs logiques en algorithmie : `NON` devient `not`, `ET and` et `OU or`. Les valeurs booléennes sont également traduites en anglais `True` et `False`.

```
print(True and False)     # affiche False
print(1 == 1 or False)    # affiche True
print(not False)          # affiche True
```

5.6 Chaînes de caractères

Python possède une multitude d'opérations sur les chaînes de caractères. Nous ne décrivons ici que les principales et nous renvoyons le lecteur à la documentation officielle de Python pour plus d'informations.

Soient deux chaînes `s` et `t`, un caractère `x` et un entier `n` :

- `x in s` : teste si `x` appartient à `s`.
- `x not in s` : teste si `x` n'appartient pas à `s`.
- `s + t` : concaténation de `s` et `t`.
- `s * n` ou `n * s` : concaténation de `n` copies de `s`.
- `len(s)` : retourne un entier représentant la longueur de `s`.
- `s.count(x` : retourne un entier représentant le nombre d'occurrences de `x` dans `s`.
- `s.index(x)` : retourne un entier représentant l'indice de `x` dans `s`.

```
s = "hello world"
print(len(s))          # 11
print(s.index('h'))      # 0
print(s.count('o'))    # 2
res = s + " coucou"    # res vaut hello world coucou
res = s * 2            # res vaut hello worldhello world
```

Remarque

Attention, en programmation, contrairement à l'algorithmie, nous commençons à compter à 0 et non 1. Le premier caractère d'une chaîne de caractères a donc la position 0 en Python.

6. Exercices

6.1 Exercice 1

Donnez l'algorithme pour calculer le volume d'un rectangle dont la largeur et la longueur soit données par l'utilisateur.

6.2 Exercice 2

Donnez l'algorithme qui convertit un nombre entier de secondes entré par l'utilisateur en un nombre d'années, de mois, de jours, d'heures, de minutes et de secondes. Pour une raison de simplicité, nous considérons qu'une année est constitué de 352 jours et un mois de 30 jours.

6.3 Exercice 3

Donnez deux algorithmes différents qui inversent les valeurs de deux variables. Ces valeurs sont données par l'utilisateur.

6.4 Exercice 4

Écrivez le script correspondant à l'algorithme calculant le volume d'un rectangle dont la largeur et la longueur sont données par l'utilisateur.

Remarque

Pensez à tester tous les cas possibles dans vos scripts Python pour ne pas avoir de mauvaises surprises lors d'une exécution non prévue.

6.5 Exercice 5

Écrivez le script correspondant à l'algorithme qui convertit un nombre entier de secondes entré par l'utilisateur en un nombre d'années, de mois, de jours, d'heures, de minutes et de secondes.

6.6 Exercice 6

Écrivez les codes Python correspondant aux deux algorithmes d'inversion de valeurs de variables.

Chapitre 3
Conditions, tests et booléens

1. Les tests et conditions

1.1 Les conditions sont primordiales

Dans notre vie, notre comportement est dirigé par une multitude de décisions à prendre. Toujours avec l'exemple du passage piéton, nous allons traverser si le voyant piéton est vert, sinon nous allons attendre **s'il** est rouge. Il en va de même pour un algorithme et un programme, nous devons guider l'ordinateur pour qu'il puisse prendre les bonnes décisions et donc exécuter correctement nos instructions.

Souvenez-vous que nous devons tout expliquer à la machine, elle ne prendra jamais une décision par elle-même, sauf peut-être le fait de s'éteindre en cas de court-circuit. Vous devez indiquer à l'ordinateur dans quels cas il peut exécuter vos instructions. Par exemple comme quand un adulte apprend à un jeune enfant à dessiner : l'adulte lui montre comment tenir le crayon, quel est le bout qui permet de dessiner, qu'on ne peut dessiner que sur une feuille et non sur une table ou un mur, etc. L'avantage de la programmation pour nous est que nos explications sont beaucoup plus simples à formuler et que la machine nous écoute forcément sans jamais en faire à sa tête et surtout comprend parfaitement du premier coup.

Les tests et conditions représentent une idée de base très simple pour bien guider notre programme : choisir d'exécuter telle ou telle instruction selon la validité d'une condition. Nous testons donc la validité d'une condition pour donner l'autorisation ou non de continuer le déroulement du programme comme : **si** le feu piéton est vert (condition), **alors** je traverse (instruction).

Le test peut également contenir une ou plusieurs alternatives : **si** le feu piéton est vert (condition), **alors** je traverse (instruction) **sinon** j'attends que le feu piéton passe au vert.

Qui dit condition, dit booléen. Les booléens sont les types les plus simples en informatique. Ils ne peuvent prendre que deux valeurs : `VRAI` ou `FAUX`. La relation entre une condition et un booléen est donc totalement explicite car une condition est toujours une expression de type booléenne.

Une condition est systématiquement le résultat d'une comparaison ou de plusieurs comparaisons reliées entre elles par les opérateurs logiques (`ET`, `OU`, `NON`).

Commençons par étudier les structures conditionnelles avec une seule comparaison pour introduire par la suite la logique booléenne (ou algèbre de Boole) pour gérer des tests avec plusieurs conditions.

1.2 Structures conditionnelles

En algorithmie, il existe deux structures conditionnelles :

- Le **SI ALORS SINON** permet de tester n'importe quelle condition avec, ou non, une ou plusieurs alternatives.
- Le **CAS PARMI,** lui, ne permet que de tester plusieurs conditions d'égalité avec une même variable en une seule instruction.

1.2.1 SI ALORS SINON

Le SI algorithmique est un bloc d'instructions soumises à une condition pour être exécutées. De ce fait, toutes les lignes comprises dans le SI doivent être **indentées** avec une nouvelle tabulation.

Remarque

Nous rappelons l'importance de l'indentation d'un algorithme ou d'un programme. L'indentation permet d'avoir une lecture simplifiée car nous pouvons voir sans réfléchir où commence un bloc et où il finit. Pour le moment, vous ne pouvez pas vraiment vous rendre compte de son importance, mais dans la suite de cet ouvrage, cela deviendra plus que pertinent avec nos premiers programmes complexes et complets.

Pour indiquer les instructions à exécuter si la condition se vérifie, nous devons les faire précéder du mot-clé ALORS. Le déroulement de l'algorithme reprend normalement, sans test donc, après le mot-clé FINSI.

Voici la syntaxe du SI ALORS :

```
SI (conditionnelle)
ALOR
   ...      // instructions à exécuter si la conditionnelle
   ...      // est vraie
FINSI
```

Exemple :

```
PROGRAMME Entier_positif
VAR
   x : ENTIER
DEBUT
   ECRIRE("Entrez un entier de votre choix")
   x <- LIRE()
   SI x > 0
   ALORS
      ECRIRE("Votre entier est positif")
   FINSI
FIN
```

Dans l'algorithme précédent, nous testons si un entier entré par l'utilisateur est positif. Que se passe-t-il si nous voulons également tester si l'entier est négatif ? Votre première pensée serait d'ajouter un nouveau SI.

Avec deux SI à la suite, l'algorithme teste quoiqu'il advienne les deux conditions, si l'entier est positif puis si l'entier est négatif, ou inversement selon l'ordre des deux SI.

Cependant, nous sommes d'accord, un entier ne peut pas être positif et négatif à la fois. Ajoutons donc simplement un SINON à notre algorithme pour les négatifs :

```
PROGRAMME Entier_positif_negatif
VAR
   x : ENTIER
DEBUT
   ECRIRE("Entrez un entier de votre choix")
   x <- LIRE()
   SI x > 0
   ALORS
      ECRIRE("Votre entier est positif")
   SINON
      ECRIRE("Votre entier est négatif")
   FINSI
FIN
```

Notre algorithme devient de plus en plus juste mais il nous reste un point à définir : l'entier valant zéro. Zéro n'est ni positif ni négatif, c'est donc un cas particulier.

Nous pouvons ajouter un SI au début de l'algorithme pour tester la valeur zéro mais cette solution n'est pas optimale. Effectivement, quelle que soit la valeur de l'entier, il sera testé deux fois : une pour l'égalité et une pour la supériorité à cause des SI qui se suivent.

Une solution propre et optimisée est de définir le zéro comme cas par défaut du SINON en y ajoutant un nouveau SI, testant si l'entier est négatif SINON c'est qu'il est à zéro (ni positif ni négatif). Mettre un SI dans SI est appelé imbriquer des instructions ou **tests imbriqués**.

```
PROGRAMME Entier_positif_negatif_nul
VAR
   x : ENTIER
DEBUT
   ECRIRE("Entrez un entier de votre choix")
   x <- LIRE()
   SI x > 0
```

```
   ALORS
      ECRIRE("Votre entier est positif")
   SINON
      SI x < 0
      ALORS
         ECRIRE("Votre entier est négatif")
      SINON
         ECRIRE("Votre entier est nul")
      FINSI
   FINSI
FIN
```

Remarque

Pensez toujours à limiter vos instructions et vos variables dans vos algorithmes et programmes afin d'optimiser la gestion de la mémoire pour les raisons évoquées au chapitre précédent et donc de rendre vos codes plus performants.

1.2.2 CAS PARMI

Lorsque nous imbriquons beaucoup de `SI`, notre algorithme peut devenir difficile à lire, donc à comprendre, donc à corriger en cas d'erreur. Une alternative possible est d'utiliser le `CAS PARMI`. Cette structure conditionnelle permet de tester la valeur d'une variable et de la comparer, **en termes d'égalité uniquement**, à plusieurs autres valeurs. Comme le `SI`, elle donne la possibilité d'avoir un cas par défaut si aucune des valeurs testées n'est correcte. En voici la syntaxe :

```
CAS variable PARMI :
   CAS1 : valeur1
      ...   // instructions à réaliser si variable vaut valeur1
   CAS2 : valeur2
      ...   // instructions à réaliser si variable vaut valeur2
   ...
   PARDEFAUT
      ...    // instructions à réaliser par défaut. Optionnel
FINCASPARMI
```

Vous pouvez tester autant de cas que nécessaire. Le cas par défaut est tout à fait facultatif.

Écrivons un algorithme permettant d'afficher le nom du mois en fonction de son numéro donné par l'utilisateur. Pour des raisons de lecture, nous nous limiterons aux six premiers mois de l'année. Notre première version sera écrite uniquement avec des `SI` et la deuxième avec le `CAS PARMI`.

```
PROGRAMME Mois_si_imbriques
VAR
   mois : ENTIER
DEBUT
   ECRIRE("Entrer un chiffre entre 1 et 6 compris")
   mois <- LIRE()
   SI mois = 1
   ALORS
      ECRIRE("Janvier")
   SINON
      SI mois = 2
      ALORS
         ECRIRE("Février")
      SINON
         SI mois = 3
         ALORS
            ECRIRE("Mars")
         SINON
            SI mois = 5
            ALORS
              ECRIRE("Mai")
              SINON
              ECRIRE("Juin")
                FINSI
           FINSI
      FINSI
   FINSI
FIN
```

```
PROGRAMME Mois_cas_parmi
VAR
   mois : ENTIER
DEBUT
   ECRIRE("Entrer un chiffre entre 1 et 6 compris")
   mois <- LIRE()
   CAS mois PARMI :
      CAS : 1
         ECRIRE("Janvier")
      CAS : 2
         ECRIRE("Février")
      CAS : 3
         ECRIRE("Mars")
      CAS : 4
         ECRIRE("Avril")
      CAS : 5
         ECRIRE("Mai")
      PARDEFAUT
         ECRIRE("Juin")
   FINCASPARMI
FIN
```

Nous remarquons facilement, que ce soit à l'écriture ou à la lecture, que l'algorithme utilisant des `SI` devient très rapidement complexe et peut nous inciter à une erreur, d'indentation ou de syntaxe par exemple. Avec l'algorithme utilisant le `CAS PARMI`, l'écriture et la lecture sont vraiment naturelles car la syntaxe est plus simple. Cependant, n'oubliez pas que le `CAS PARMI` ne peut que tester des égalités alors que `SI` peut tester tout type de comparaison.

Regardons maintenant comment combiner plusieurs tests dans une structure conditionnelle.

2. La logique booléenne

2.1 Conditions multiples

Écrire une condition qui teste la validité d'un seul fait est assez logique, voire simple. Les conditions augmentent en complexité dès lors que nous augmentons le nombre de fais à valider, que ce soit dans la vie de tous les jours ou en informatique.

Lorsque nous testons des conditions multiples en tant qu'être humain, notre cerveau raisonne tellement vite que nous n'avons pas l'impression de réfléchir ni même de résoudre une équation. Et pourtant...

Reprenons l'exemple du passage piéton. Il vous paraît normal, avant de traverser à un feu, de vérifier que le voyant piéton est vert et d'également vérifier qu'aucune voiture ne grille le feu. Vous analysez donc deux conditions et avez l'impression de les faire en même temps avec une seule et unique condition. Mais votre cerveau reçoit bien deux conditions à vérifier, donc il résout ces deux conditions dans une équation.

Comme nous l'avons indiqué dans le chapitre d'introduction de cet ouvrage, vous devez tout décrire à la machine, faire du vrai pas-à-pas, aucun raccourci n'est possible. Il nous faut donc une manière simple de représenter les conditions multiples.

Pour formuler correctement cette équation avec plusieurs conditions, George Boole, un mathématicien du XIX^{e} siècle, créa une algèbre binaire que Claude Shannon utilisa plus d'un siècle plus tard en informatique. L'algèbre de Boole, appelée aussi logique booléenne selon le contexte, permet d'analyser plusieurs conditions en même temps dans une même équation, donc dans une même structure conditionnelle. Vous comprenez maintenant d'où vient le nom du type booléen.

L'implémentation de cette algèbre est quasiment naturelle en informatique. Un ordinateur fonctionnant par impulsion électrique, le `FAUX` est donc représenté par le 0 ou l'absence de courant, le `VRAI` par le 1 ou la présence d'un courant.

Ce procédé de calcul donne priorité à certaines conditions par rapport à d'autres en fonction des opérateurs choisis.

Nous renvoyons le lecteur souhaitant approfondir ce sujet vers des ouvrages traitant de l'architecture de l'ordinateur, par exemple *Le langage assembleur* écrit par Olivier Cauet et publié aux Éditions ENI.

2.2 Algèbre ou logique de Boole

La logique booléenne repose sur trois opérateurs : le `ET`, le `NON` et le `OU`, et deux valeurs : `VRAI` et `FAUX`.

Avec ces trois opérateurs logiques, nous allons pouvoir indiquer à l'ordinateur comment résoudre correctement des conditions multiples.

2.2.1 ET logique

Le `ET` logique représente la **multiplication** de deux booléens, donc de deux conditions, donc de deux comparaisons en informatique. Il faut que les deux booléens du `ET` soient à `VRAI` pour que le tout soit `VRAI`. Par exemple, il faut que le feu piéton soit au vert et que le passage soit libre pour traverser.

Remarque

Le ET logique est plus précisément une analogie du signe de la multiplication de deux nombres : si les deux nombres sont positifs, le résultat sera positif sinon il sera négatif.

La figure ci-dessous représente la table de vérité du `ET` logique, c'est-à-dire tous les résultats possibles en fonction des valeurs des deux booléens opérandes, `a` et `b`, de cet opérateur.

a/b	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

Table de vérité du ET logique

Grâce à cette table de vérité, nous pouvons poser notre condition multiple à l'ordinateur pour qu'il la résolve :

```
PROGRAMME Et_logique
VAR
feu_pieton_vert : BOOLEEN
passage_libre : BOOLEEN
DEBUT
   ECRICRE("Le feu est-il vert ? VRAI ou FAUX")
   feu_pieton_vert <- LIRE()
   ECRICRE("Le passage est-il libre ? VRAI ou FAUX ")
   passage_libre <- LIRE()
   SI feu_pieton_vert ET passage_libre
   ALORS
      ECRIRE("Vous pouvez traverser")
   SINON
      ECRIRE("Vous ne devez pas traverser")
   FINSI
FIN
```

2.2.2 OU logique

Le OU logique est la représentation de l'addition en mathématiques classiques. Il suffit que l'un des deux opérandes booléens soit VRAI pour que le résultat soit VRAI, comme le montre la table de vérité du OU dans la figure ci-dessous.

a/b	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

Table de vérité du OU logique

Prenons l'exemple d'un repas. Que nous buvions ou que nous mangions, nous nous restaurons.

```
PROGRAMME Se_restaurer
VAR
   boire : BOOLEEN
   manger : BOOLEEN
DEBUT
   ECRICRE("Buvez-vous ?")
   boire <- LIRE()
```

```
    ECRICRE("Mangez-vous ?")
    manger <- LIRE()
    SI boire OU manger
    ALORS
       ECRIRE("Vous vous restaurez")
    SINON
       ECRIRE("Vous jeûnez")
    FINSI
FIN
```

2.2.3 NON logique

Le dernier opérateur est un opérateur unaire, il ne fonctionne qu'avec une valeur. Le NON logique inverse la valeur du booléen sur lequel il s'applique, le VRAI devient FAUX et inversement, comme indiqué dans sa table de vérité dans la figure ci-dessous. Nous pouvons vulgariser cet opérateur en le comparant à un esprit de contradiction.

a	NON a
VRAI	FAUX
FAUX	VRAI

Table de vérité du NON logique

Voici un algorithme simple sur l'utilisation du NON :

```
PROGRAMME Non_logique
VAR
   b : BOOLEEN <- VRAI
DEBUT
   ECRIRE(NON b)          // affiche FAUX
   ECRIRE (NON (NON b))   // Affiche VRAI
FIN
```

2.2.4 Règles de priorités

Le ET logique est prioritaire sur le OU logique, tout comme la multiplication est prioritaire sur l'addition. Le NON logique est prioritaire sur le ET logique donc sur le OU logique également. Nous pouvons résumer ces règles par : NON est prioritaire sur le ET qui est prioritaire sur le OU.

Ces deux règles de priorités sont tout à fait identiques à la pensée humaine. La négative, c'est-à-dire le NON logique, s'applique avant tout le reste, comme en français, puis la conjonction (l'union, c'est-à-dire le ET logique) puis la disjonction (l'alternative, c'est-à-dire le OU logique).

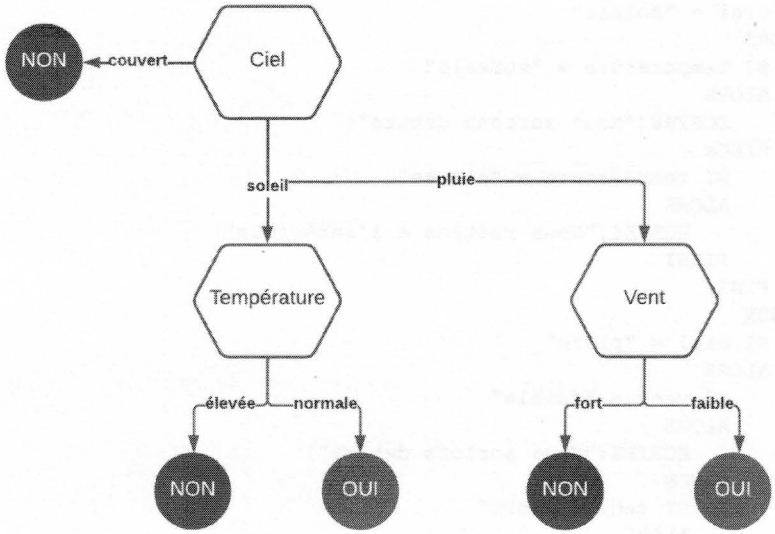
Si nous voulons casser ces deux règles, nous pouvons utiliser les parenthèses comme en mathématiques classiques car elles permettent de court-circuiter les priorités de notre calcul avec des sous-calculs à évaluer en premier.

Jouons un peu avec des calculs booléens :

- VRAI ET FAUX OU VRAI donne VRAI.
- (VRAI ET FAUX) OU VRAI donne VRAI (les parenthèses ne cassent pas l'ordre de priorité ici).
- VRAI ET (FAUX OU VRAI) donne VRAI car FAUX OU VRAI donne VRAI et est l'expression calculée en premier.
- NON VRAI ET VRAI donne FAUX car cela revient à FAUX ET VRAI, le NON logique étant prioritaire.
- NON (FAUX ET VRAI) donne VRAI car le NON logique est appliqué au résultat du ET logique, soit NON FAUX.

2.2.5 Un exemple concret

Prenons en exemple un arbre de décision assez simple : pouvons-nous sortir dehors en fonction de critères prédéfinis, résumés dans la figure ci-dessous :



Arbre de décision du cas : dois-je sortir ou non

Dans cet exemple, nous ne sortons pas si le ciel est couvert. S'il fait soleil, notre sortie dépendra de la température : si elle est normale, nous sortons, si elle est élevée nous ne sortons pas. En cas de pluie, le vent sera décisionnaire : nous sortons avec un vent faible et nous restons à l'intérieur avec un vent fort. Traduisons toute cette réflexion en algorithme.

```
PROGRAMME Sortir_ou_rester_non_optimise
VAR
   ciel : CHAINE
   temperature : CHAINE
   vent : CHAINE
DEBUT
   ECRIRE("Entrez la valeur du ciel : couvert, pluie ou soleil")
   ciel <- LIRE()
   ECRIRE("Entrez la valeur de la température : normale ou elevee")
   temperature <- LIRE()
   ECRIRE("Entrez la valeur du vent : faible ou fort")
```

```
    vent <- LIRE()
    SI ciel = "couvert"
    ALORS
       ECRIRE("Nous restons à l'intérieur")
    SINON
       SI ciel = "soleil"
       ALORS
          SI temperature = "normale"
          ALORS
             ECRIRE("Nous sortons dehors")
          SINON
             SI temperature = "elevee"
             ALORS
                ECRIRE("Nous restons à l'intérieur")
             FINSI
          FINSI
       SINON
          SI ciel = "pluie"
          ALORS
             SI vent = "faible"
             ALORS
                ECRIRE("Nous sortons dehors")
             SINON
                SI vent = "fort"
                ALORS
                   ECRIRE("Nous restons à l'intérieur")
                FINSI
             FINSI
          FINSI
       FINSI

    FINSI
FIN
```

Nous pouvons écrire nos conditions avec un seul `SI` grâce à la logique booléenne pour un algorithme plus performant.

```
PROGRAMME Sortir_ou_rester
VAR
   ciel : CHAINE
   temperature : CHAINE
   vent : CHAINE
DEBUT
   ECRIRE("Entrez la valeur du ciel : couvert, pluie ou soleil")
   ciel <- LIRE()
   ECRIRE("Entrez la valeur de la température : normale ou elevee")
```

```
    temperature <- LIRE()
    ECRIRE("Entrez la valeur du vent : faible ou fort")
    vent <- LIRE()
    SI (ciel = "soleil" ET temperature = "normale")
OU (ciel = "pluie" ET vent = "faible")
    ALORS
       ECRIRE("Nous sortons dehors")
    SINON
       ECRIRE("Nous restons à l'intérieur")
    FINSI
FIN
```

Toutes les bases théoriques sur la gestion des conditions étant posées, passons maintenant à son implémentation en Python.

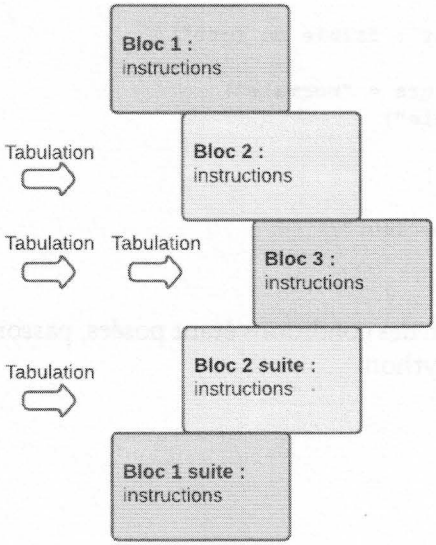
3. Les blocs en python

3.1 L'importance de l'indentation

Python est un langage d'une syntaxe assez particulière qui nécessite de la rigueur de la part du développeur. Cette rigueur permet d'alléger la syntaxe de Python et de forcer les bonnes pratiques de développement.

En effet, en Python, un code qui n'est pas indenté n'est pas exécutable par l'interpréteur.

Un script Python est décomposé en blocs. Chaque bloc doit être indenté par une nouvelle tabulation comme le montre la figure ci-dessous. Pour remonter au bloc précédent, il suffit d'enlever une tabulation aux instructions. Pour indiquer à l'interpréteur qu'un bloc commence, l'instruction doit finir par le caractère "**:**".



Indentation avec Python

La règle est simple : un bloc d'instructions = une indentation.

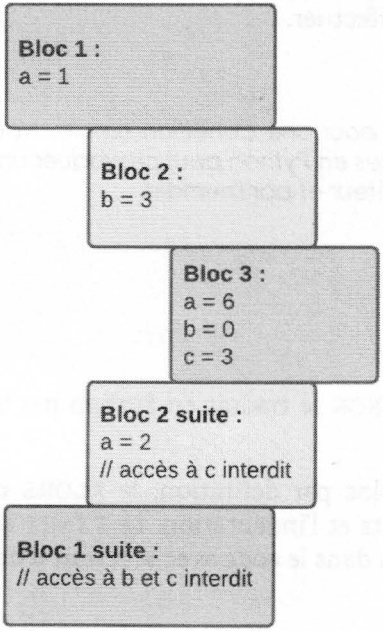
Pourquoi abordons-nous cette règle ? Tout simplement car une structure conditionnelle en Python doit être représentée par un bloc. Comme Python utilise le saut de ligne pour la fin d'une instruction, il utilise un bloc pour une instruction conditionnelle.

3.2 Visibilité des variables

Cette notion de bloc impose d'aborder la visibilité des variables, appelée encore la portée ou le scope.

Les variables d'un algorithme peuvent être utilisées dans n'importe quelle ligne car elles sont toutes connues de tout l'algorithme grâce à leur déclaration dans le bloc VAR avec le début de l'algorithme. Ces variables sont dites globales.

Les variables peuvent être déclarées n'importe quand en Python, contrairement à l'algorithmie. Les variables ne sont donc pas globales mais locales au bloc dans lequel est sont déclarés. Une variable déclarée dans le bloc 1 de la figure ci-dessous peut être accessible par l'ensemble des blocs imbriqués dans ce dernier. En revanche, une variable déclarée dans le bloc 2 n'est accessible que par celui-ci et le bloc 3. Il en va de même avec le bloc 3 : une variable déclarée dans le bloc 3 ne peut être accessible que par ce dernier. Et ainsi de suite si votre script comporte plus de blocs que notre exemple.



Variables et blocs

3.3 Conditions en Python

Comme en algorithmie, les conditions en Python doivent être de type booléen. Elles sont obtenues par une comparaison avec les opérateurs : <, <=, >, >=, == ou !=.

Pour poser des conditions multiples, vous devez utiliser les opérateurs logiques `and`, `or` ou `not`. L'ordre de priorité est le même que celui de la logique booléenne, n'hésitez donc pas à utiliser des parenthèses pour que votre condition décrive bien les tests que vous voulez effectuer.

Remarque

Attention à ne pas utiliser de parenthèses pour une condition simple. Nous verrons par la suite que l'abus de parenthèses en Python peut provoquer une confusion à la lecture du script par l'interpréteur et par l'humain.

3.4 Instructions conditionnelles

3.4.1 SI ALORS SINON

La structure conditionnelle `SI ALORS SINON` se traduit en Python par les instructions `if`, `else` et `elif`.

Une instruction conditionnelle étant un bloc par définition, le `ALORS` de l'algorithmie est remplacé par les deux points et l'indentation. Le `FINSI` est tout simplement le retour d'une indentation dans le code avec le retrait d'une tabulation.

La traduction du `SINON` est le `else`. Les tests imbriqués sont, en Python, traduits grâce au `elif`, qui est la contraction des mots *else* et *if*. Cette syntaxe allège énormément les tests imbriqués qui deviennent plus parlants et moins complexes à écrire.

Remarque

Selon vos besoins, vous pouvez également imbriquer des tests dans d'autres tests, aucune règle de syntaxe ne vous en empêche.

Comme en algorithmie, seule l'instruction `if` est obligatoire, les instructions `else` et `elif` étant optionnelles.

Traduisons les trois algorithmes testant le signe d'un entier en Python.

```
x = int(input('Entrer un entier')
if x > 0 :
   print("x est positif")
```

```
x = int(input('Entrer un entier')
if x > 0 :
   print("x est positif")
else :
   print("x est négatif)
```

```
x = int(input('Entrer un entier')
if x > 0 :
   print("x est positif")
elif x < 0 :
   print("x est négatif")
else :
   print("x est nul")
```

Avec le dernier exemple, nous nous rendons facilement compte de l'élégance et de la simplicité de l'instruction `elif`.

Terminons l'apprentissage du `SI ALORS SINON` en Python avec les deux scripts correspondant aux algorithmes qui déterminent si nous sortons ou restons à l'intérieur.

– Avec les `SI` imbriqués :

```
ciel = input("Entrez le ciel : soleil, couvert ou pluie ?")
temperature = input("Entrez la température : elevee ou normale ?")
pluie = input("Entrez le vent : fort ou faible ?")
if ciel == "couvert" :
   print("Nous restons à l'intérieur")
elif ciel == "soleil" :
   if temperature == "normale" :
      print("Nous sortons dehors")
   elif temperature == "elevee" :
      print("Nous restons à l'intérieur")
elif ciel == "pluie" :
   if vent == "faible" :
```

```
        print("Nous sortons dehors")
    elif vent == "fort" :
        print("Nous restons à l'intérieur")
```

– Avec un seul `SI` :

```
ciel = input("Entrez le ciel : soleil, couvert ou pluie ?")
temperature = input("Entrez la température : elevee ou normale ?")
pluie = input("Entrez le vent : fort ou faible ?")
if (ciel == 'soleil' and temperature == 'normale') or
(ciel == 'pluie' and vent == 'faible') :
    print("Nous sortons dehors")
else :
    print("Nous restons à l'intérieur")
```

Nous remarquons avec ce script que la complexité réside dans la création de la condition et non dans la syntaxe Python, tout comme en algorithmie.

3.4.2 Opérateur ternaire

L'opérateur ternaire est une structure conditionnelle simple qui s'écrit sur une ligne et non sur un bloc. Il est principalement utilisé pour affecter une valeur selon une condition avec obligatoirement une valeur par défaut si la condition n'est pas validée. Voici sa syntaxe :

```
expression1 if condition else expression2
```

L'opérateur ternaire retourne l'`expression1` si la `condition` se révèle être vraie sinon il retourne l'`expression2`.

Voyons cet opérateur avec un cas concret. Nous allons demander son âge à l'utilisateur et lui dire s'il est majeur (plus de 18 ans) ou mineur (moins de 18 ans).

```
age = int(input("Entrer votre âge")
res = 'mineur' if age < 18 else 'majeur'
print("Vous êtes", res)
```

3.4.3 CAS PARMI

Vu la simplicité qu'apporte l'instruction `elif`, la structure conditionnelle `CAS PARMI` n'existe pas en Python : nous devons toujours utiliser un `if`.

Les deux algorithmes qui affichent le nom du mois en fonction de son numéro deviennent donc le seul script suivant :

```
mois = int(input('Entrer un entier entre 1 et 6'))
if mois == 1 :
   print('Janvier')
elif mois == 2 :
   print('Février')
elif mois == 3 :
   print('Mars')
elif mois == 4 :
   print('Avril')
elif mois == 5 :
   print('Mai')
else :
   print('Juin')
```

Cet exemple illustre bien la puissance de l'instruction `elif` et le fait que le `CAS PARMI` ne soit plus utile en Python.

4. Exercices

4.1 Exercice 1

Écrivez l'algorithme puis le script Python calculent la remise suivante à un montant de type réel entré par l'utilisateur : il est accordé une remise de 5 % pour tout montant compris entre 100 et 500 € et 8 % au-delà. Pensez à tester votre script avec tous les cas possibles.

4.2 Exercice 2

Écrivez l'algorithme puis le script Python faisant saisir à l'utilisateur trois entiers, i, j et k, et les triant par ordre croissant et les afficher pour vérifier votre tri.

4.3 Exercice 3

Écrivez l'algorithme puis le script Python qui calcule le signe du produit de deux réels entrés par l'utilisateur sans utiliser la multiplication ni aucun autre calcul.

4.4 Exercice 4

Écrivez un algorithme qui détermine si une année entrée par l'utilisateur est bissextile. Une année bissextile est un entier divisible par quatre seulement s'il ne représente pas une année de centenaire (2000, 1900, 1800, etc.). Dans ce cas, l'entier doit être également divisible par 400. Codez le script Python correspondant.

Chapitre 4
Les boucles

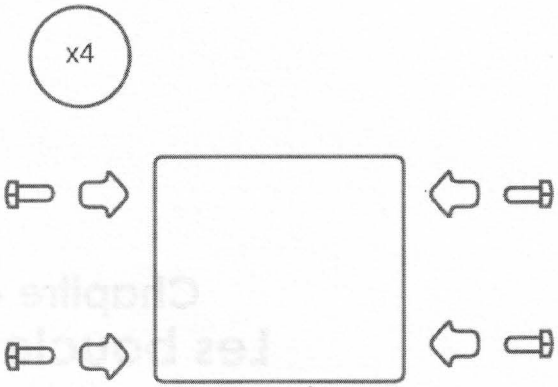
1. Les structures itératives

1.1 Itérer pour mieux programmer

Dans le chapitre précédent, nous avons appris à guider notre algorithme en lui demandant de tester des conditions afin d'exécuter ou non des instructions. Ce fut le premier pas d'une réflexion importante pour communiquer quoi faire et surtout quand le faire à la machine. Passons maintenant au deuxième pas primordial : les itérations.

Reprenons les deux exemples du premier chapitre : la notice de montage d'un meuble en kit et la recette de la tarte aux pommes.

Dans le cas d'une notice de montage, il est souvent demandé de répéter les mêmes actions sur les mêmes types de pièces. Un exemple est représenté par la figure suivante : il nous est demandé d'insérer quatre vis dans une planche et de le faire pour les quatre planches. Il s'agit bien des mêmes actions à faire quatre fois, donc une itération de quatre tours. Le x4 de cette figure indique ce nombre d'itérations et nous semble parlant et tout à fait logique. Nous allons découvrir comment le traduire pour l'ordinateur dans la suite de ce chapitre.



Une étape d'une notice de montage de meuble en kit

Pour la recette de cuisine, les répétitions se trouvent dans la gestion des pommes : pour chaque pomme, l'éplucher et la couper en tranches qui seront disposées dans la tarte. Nous n'arrêtons pas ces actions tant qu'il y a encore des pommes pour la tarte. Nous pouvons également dire que nous continuons ces actions jusqu'à qu'il ne reste plus de pommes.

À travers ces deux exemples, nous avons vu les trois grands cas d'itérations possibles en informatique : avec **un nombre fixe d'itérations** et avec le **test de la validité d'une condition**, avant ou après la répétition des actions.

L'ordinateur, tout comme nous, est amené à exécuter plusieurs fois des actions identiques, ce que vont permettre les **instructions itératives**, souvent appelées **boucles** par les développeurs. L'itération représente ici une séquence d'instructions à exécuter plusieurs fois, avec un nombre fixe de répétitions ou une condition à valider pour arrêter ou continuer les itérations.

1.2 Comment itérer proprement ?

L'algorithmie et les langages de programmation possèdent tous plusieurs instructions pour faire des itérations. Chacune de ces instructions correspond à une itération particulière, comme nous l'avons vu avec les exemples de la section précédente :

- Nous voulons continuer à répéter des instructions tant qu'une instruction est validée : `TANT QUE` (recette de la tarte, première remarque).
- Nous voulons continuer à répéter des instructions jusqu'à ce qu'une condition soit validée mais en forçant la première itération : `REPETER JUSQU'A` (recette de la tarte, deuxième remarque).
- Nous savons exactement combien de répétitions nous devons exécuter : `POUR` (notice de montage).

Remarque

Les structures itératives sont également présentes pour éviter le copier-coller d'instructions. En effet, si vous devez copier un certain nombre d'instructions identiques et les coller les unes après les autres, cela peut vouloir dire que vous devez utiliser une structure itérative. Par exemple, pour afficher un compte à rebours de 10 à 0, il est plus simple de boucler sur la décrémentation d'un entier et son affichage que de réécrire onze fois sa décrémentation et son affichage.

2. Tant Que

2.1 Principe et syntaxe

Nous pouvons comparer le `TANT QUE` à un `SI ALORS` qui se répète tant que la condition est valide. Cette structure réalise un nombre indéfini d'itérations qui va dépendre de la validité de la condition. Ces itérations ne s'arrêtent que lorsque la condition n'est plus validée.

Les instructions à répéter se retrouvent dans le bloc de cette structure :

```
TANT QUE (conditionnelle)
FAIRE
   ...    // suite d'instructions à répéter
FINTANTQUE
```

Revenons en cuisine avec notre tarte aux pommes. Une manière de gérer les pommes avec cette structure serait la suivante :

```
TANT QUE il reste des pommes
FAIRE
   Prendre une pomme
   Éplucher la pomme
   Couper la pomme en tranches
FINTANTQUE
```

La difficulté du TANT QUE est de trouver la bonne condition à tester. En effet, si cette condition est toujours satisfaite, votre itération sera infinie, elle se n'arrêtera jamais et votre programme restera bloqué sur les instructions du bloc, sans jamais se terminer. Si la condition n'est jamais satisfaite, la boucle ne s'exécute pas une seule fois. C'est ce que les développeurs appellent une **boucle infinie**.

Regardons maintenant le TANT QUE dans des algorithmes concrets.

2.2 Exemples

Un exemple simple du TANT QUE est de faire compter l'ordinateur. L'algorithme arrêtera d'afficher le compteur uniquement lorsque ce dernier aura atteint la valeur entrée par l'utilisateur.

```
PROGRAMME Comptage_tant_que
VAR
   cpt : ENTIER <- 1
   borne : ENTIER
DEBUT
   ECRIRE("Entrez un nombre supérieur à 1")
   borne <- LIRE()
   TANT QUE cpt ? borne
   FAIRE
      ECRIRE(cpt)
```

```
      cpt <- cpt + 1
   FINTANTQUE
FIN
```

Un autre exemple un peu plus mathématique : donner la valeur entière d'un réel positif. Pour ce faire, la logique est assez simple, nous allons **incrémenter** un entier (l'augmenter de 1) tant qu'il est plus petit que la valeur du réel **décrémenté** de 1 (diminué de 1).

```
PROGRAMME Partie_entiere
VAR
   partie_entiere : ENTIER <- 0
   x : REEL
DEBUT
   ECRIRE("Entrez un nombre réel positif ou nul")
   x <- LIRE()
   TANT QUE (partie_entiere + 1) <= x
   FAIRE
      partie_entiere <- partie_entiere + 1
   FINTANTQUE
   ECRIRE("La partie entière de ", x, " est ", partie_entiere)
FIN
```

Un dernier exemple est un menu utilisateur qui se répète à moins que l'utilisateur entre une chaîne de caractères précise. Dans notre cas, nous allons faire un menu simpliste qui demande à l'utilisateur de continuer (entrée égale à oui) ou d'arrêter (entrée égale à non).

```
PROGRAMME Menu_tant_que
VAR
   reponse : CHAINE
DEBUT
   ECRIRE("Voulez-vous continuer? (oui/non)")
   reponse <- LIRE()
   TANT QUE reponse = "oui"
   FAIRE
      ECRIRE("Voulez-vous continuer? (oui/non)")
      reponse <- LIRE()
   FINTANTQUE
FIN
```

Ce dernier exemple ne vous pose-t-il pas un problème ? En effet, nous avons les deux instructions ECRIRE et LIRE qui sont écrites une première fois avant la structure itérative et qui se répètent une fois de plus dans le bloc de cette dernière. Ce n'est pas optimal car nous écrivons plusieurs fois les mêmes instructions sans un grand intérêt. Pour corriger ce défaut, nous allons voir maintenant la structure itérative REPETER JUSQU'A.

3. Répéter ... Jusqu'à

3.1 Principe et syntaxe

Le REPETER JUSQU'A peut être considéré comme un TANT QUE **inversé** : les instructions sont d'abord répétées une fois avant de tester la condition. Quelle que soit la validité de la condition, les instructions du bloc sont forcément exécutées une fois.

```
REPETER
   ...   // suite d'instructions à répéter
JUSQU'A (conditionnelle)
```

Notre deuxième version pour la gestion des pommes de notre tarte est donc la suivante :

```
REPETER
FAIRE
   Prendre une pomme
   Éplucher la pomme
   Couper la pomme en tranches
JUSQU'A il ne reste plus de pommes
```

Remarque

Nous pouvons remarquer que les conditions du TANT QUE et du REPETER JUSQU'A sont des ***conditions inverses****. Nous testons s'il reste des pommes avec le TANTQUE alors que nous testons s'il ne reste plus de pommes avec le REPETER JUSQU'A.*

3.2 Exemple

Corrigeons notre menu avec cette nouvelle structure itérative :

```
PROGRAMME Menu
VAR
   reponse : CHAINE
DEBUT
   REPETER
      ECRIRE("Voulez-vous continuer? (oui/non)")
      reponse <- LIRE()
   JUSQU'A reponse = non
FIN
```

Nous remarquons bien avec cet algorithme que le `REPETER JUSQU'A` est plus approprié à notre menu utilisateur car les opérateurs `LIRE` et `ECRIRE` ne sont plus écrits qu'une seule fois avec le même résultat que l'algorithme précédent.

4. Pour

4.1 Principe et syntaxe

La structure itérative `POUR` correspond au cas de la répétition de la notice de montage du meuble en kit. Le bloc d'instruction est **répété un nombre de fois précis**, déterminé avant la structure.

Le `POUR` doit toujours être utilisé avec une variable qui nous permet de savoir à quel numéro d'itération nous sommes actuellement : le **compteur**, communément nommé **i** en programmation.

Ce compteur doit partir d'une valeur initiale pour arriver à une valeur finale. Lorsque ce dernier dépasse cette valeur finale, les itérations cessent et la suite du programme est exécutée.

Pour aller de la valeur initiale à la valeur finale, vous devez déclarer le **pas** : comment le compteur va changer de valeur d'une itération à l'autre. Ce pas est appliqué à chaque tour et doit être un entier, positif ou négatif. La valeur du pas est additionnée automatiquement au compteur à chaque nouvelle itération.

Remarque

Si vous voulez écrire un POUR décroissant, qui décrémente donc le compteur, il vous suffit de déclarer un pas négatif.

```
VAR compteur : ENTIER
POUR compteur ALLANT de ... à ... AU PAS DE ...
FAIRE
   ...   // suite d'instructions à répéter
FINPOUR
```

Modélisons la notice de montage en un algorithme :

```
PROGRAMME Notice
   compteur : ENTIER
DEBUT
   POUR compteur ALLANT de 1 à 4 AU PAS DE 1
   FAIRE
      Prendre une planche sans vis
      Mettre les quatre vis dans cette planche
   FINPOUR
FIN
```

4.2 Exemples

L'exemple le plus classique du POUR est la table de multiplication. Nous allons afficher à l'utilisateur la table de multiplication de son choix.

```
PROGRAMME Table_multiplication
   i : ENTIER
   entree_utilisateur : ENTIER
DEBUT
   ECRIRE("Quelle table de multiplication voulez-vous afficher ?
(entier entre 1 et 10)")
   entree_utilisateur <- LIRE()
   POUR i ALLANT de 1 à 10 AU PAS DE 1
   FAIRE
      ECRIRE(i," x ", entree_utilisateur," = ",
entree_utilisateur x i)
   FINPOUR
FIN
```

Maintenant que vous connaissez le POUR, vous devez vous poser des questions sur notre algorithme Comptage_tant_que. Vous paraît-il non approprié à la logique ? La réponse est évidemment oui : qui dit compter jusqu'à une fin donnée, dit un compteur, donc une structure POUR.

```
PROGRAMME Comptage
VAR
   cpt : ENTIER <- 1
   borne : ENTIER
DEBUT
   ECRIRE("Entrez un nombre supérieur à 1")
   borne <- LIRE()
   POUR i ALLANT de 1 à borne AU PAS DE 1
   FAIRE
      ECRIRE(cpt)
   FINPOUR
FIN
```

Cet algorithme est plus sûr que celui utilisant le TANT QUE : l'incrémentation du compteur étant gérée automatiquement par le POUR, nous sommes sûrs que cette boucle se terminera alors que le TANT QUE laisse à la charge du développeur cette incrémentation, ce qu'il risque d'oublier, d'où le risque de boucle infinie.

5. Structures itératives imbriquées

Nous avons un peu trop vulgarisé nos deux algorithmes de la vie courante. En effet, il y a un traitement à faire pour chaque tranche de pomme dans la recette, et un autre pour chaque vis dans la notice de montage. Nous avons donc besoin d'une itération dans une autre.

Comme pour la structure conditionnelle SI, vous pouvez tout à fait imbriquer des structures itératives les unes dans les autres et imbriquer des conditionnelles dans des itératives, et vice versa. Les possibilités d'imbrication sont infinies, ce qui nous permet d'écrire des programmes complexes intéressants pour l'utilisateur.

Détaillons maintenant nos deux exemples en imbriquant une structure itérative dans celles qui existent déjà :

```
PROGRAMME recette_tarte_pommes
TANT QUE il y a des pommes
FAIRE
   Prendre une pomme
   Eplucher la pomme
   Couper la pomme en tranchess
   REPETER
      Disposer la tranche dans la tarte
   JUSQU'A il ne reste plus de tranches
FINTANTQUE
```

```
PROGRAMME Notice_complete
   i, j : ENTIER
DEBUT
   POUR i ALLANT de 1 à 4 AU PAS DE 1
   FAIRE
      POUR j ALLANT de 1 à 4 AU PAS DE 1
      FAIRE
         Mettre la i ème vis dans la j ème planche
      INPOUR
   FINPOUR
FIN
```

Remarque

Pour les structures `POUR` imbriquées, il est de convention d'utiliser les lettres i, j, k, l... pour les compteurs, comme en mathématiques.

Souvenez-vous des cahiers de brouillon, plus précisément du dos des cahiers : les tables de multiplication. Nous allons écrire l'algorithme qui permet de générer ces tables.

```
PROGRAMME Tables_multiplication
   i, j : ENTIER
DEBUT
   POUR i ALLANT de 1 à 10 AU PAS DE 1
   FAIRE
      ECRIRE("Table de ", i)
      POUR j ALLANT de 1 à 10 AU PAS DE 1
      FAIRE
         ECRIRE(i," x ",j," = ",j x i)
      FINPOUR
   FINPOUR
FIN
```

6. Attention danger

Le premier danger des structures conditionnelles est la **boucle infinie**. Vous devez être certain que votre condition de sortie sera validée au cours de l'exécution de l'algorithme ou que votre compteur atteindra la borne maximale.

Prenons deux exemples de boucles infinies.

Le premier est un comptage avec la boucle `TANT QUE` :

```
PROGRAMME Comptage_infini
VAR
   cpt : ENTIER <- 1
   borne : ENTIER
DEBUT
   ECRIRE("Entrez un nombre supérieur à 1")
   borne <- LIRE()
   TANT QUE cpt ? borne
   FAIRE
      ECRIRE(cpt)
   FINTANTQUE
FIN
```

Cet algorithme n'augmente jamais la valeur de la variable `cpt`. De ce fait, la valeur de cette variable sera toujours différente de celle de la variable borne et les itérations ne s'arrêteront jamais, 1 sera donc affiché à l'infini.

Remarque

Si vous avez un nombre d'itérations fixes, utilisez toujours une structure `POUR`.

Le second est une légère modification de notre table de multiplication :

```
PROGRAMME Table_multiplication_infini
   i : ENTIER
   entree_utilisateur : ENTIER
DEBUT
   ECRIRE("Quelle table voulez-vous afficher?")
   entree_utilisateur <- LIRE()
   POUR i ALLANT de 1 à 10 AU PAS DE -1
   FAIRE
      ECRIRE(i," x ", entree_utilisateur," = ", entree_utilisateur x i)
   FINPOUR
FIN
```

Nous avons glissé une petite erreur de logique dans cet algorithme : le pas du compteur est de -1 et non de 1. Le compteur aura donc comme valeur 1, 0, -1, -2, -3, ... et n'arrivera jamais à la valeur 10. La boucle sera donc infinie, affichant à l'écran pour une entrée utilisateur de 10 : 10, 0, -10, -20, -30...

Pensez donc bien à bien choisir les éléments de vos structures itératives pour éviter les boucles infinies.

Un dernier danger vient de l'imbrication de plusieurs structures itératives.

Plus vous allez imbriquer de structures les unes dans les autres, plus les opérations vont être répétées et donc plus l'ordinateur devra retenir ces instructions, les résultats intermédiaires et les valeurs des compteurs entre autres. Selon l'objectif de ces imbrications, par exemple un calcul complexe avec six imbrications de POUR itérant 1000 fois chacune, vous pouvez saturer la mémoire de l'ordinateur. Une fois saturée, le programme ne pourra plus fonctionner, vous aurez atteint les limites de la machine. Ce problème est appelé **dépassement de mémoire** et vous ne pourrez rien y faire sauf revoir votre logique.

7. Itérons en python

7.1 Pour

L'une des particularités du langage Python est la structure POUR. En algorithmie, cette structure a besoin d'un compteur, d'une valeur initiale et d'une valeur finale de ce compteur et d'un pas. Python traduit cette structure par POUR CHAQUE.

Le POUR CHAQUE de Python itère sur un ensemble de valeurs et va mettre dans une variable chacune de ces valeurs, une par itération.

Sa syntaxe est la suivante :

```
for variable in ensemble_valeurs :
    # bloc d'instructions à répéter
```

Commençons par itérer sur une chaîne de caractères (qui est bien un ensemble de caractères) :.

```
for carac in "hello world" :
    print(carac, end="")
print()  # un saut de ligne en plus pour l'affichage
```

Le `for` précédent affiche chaque caractère de la chaîne *hello world* sur une ligne, les uns après les autres.

Comment faire pour utiliser POUR CHAQUE afin d'écrire notre POUR ? Pour ce faire, il existe une instruction particulière en Python donnant un ensemble de valeurs numériques : le `range`.

Le `range` prend a minima en paramètre la borne maximale de l'ensemble. `range(10)` va nous donner les dix premiers entiers. Cette borne n'est pas comprise dans l'ensemble de valeurs retournées.

Par défaut, le `range` commence avec la valeur initiale 0. Si vous voulez commencer avec une autre valeur initiale, il faut l'indiquer en premier paramètre : `range(1,11)` retourne les 10 premiers entiers positifs.

Pour changer le pas qui est de 1 par défaut, il vous faut ajouter un troisième paramètre : `range (1, 11, 2)` donne les cinq premiers entiers impairs.

Utilisons donc ce range pour écrire en Python notre structure itérative POUR.

```
for i in range(10) :
   print(i, end ='')    # affiche 0123456789
print()          # juste un saut de ligne
for i in range(1, 11) :
   print(i, end ='')    # affiche 12345678910
print()
for i in range(1, 11, 2) :
   print(i, end ='')    # affiche 13579
print()
```

Nous retrouvons bien notre structure itérative POUR de l'algorithmie avec une syntaxe moins verbeuse : par défaut, le compteur commence à 1 et le pas est de 1.

Remarque

Par convention, en programmation, les bornes maximales ne sont jamais incluses, sauf quelques rares exceptions.

Nous pouvons traduire notre algorithme calculant une table de multiplication :

```
table = int(input("Entrer un entier compris entre 1 et 10"))
for i in range(1, 11) :
   print(i, 'x', table, '=', table*i)
```

7.2 Tant que

Le Python ne change que la philosophie du POUR, le TANT QUE reste le même, il est juste traduit en anglais par le mot-clé `while`.

Sa syntaxe est la suivante :

```
while conditionnelle :
   # bloc d'instructions à répéter
```

Traduisons notre algorithme `Partie_entiere` en Python :

```
x = eval(input("Entrer un réel positif ou nul"))
n = 0
m = n + 1
while m <= x :
   n += 1
   m = n + 1
print("la partie entière de", x, "est", n)
```

7.3 Répéter jusqu'à

La structure itérative `REPETER JUSQU'A` n'existe pas en Python. Il nous faudra utiliser une instruction `while` à la place, quitte à répéter des instructions :

```
reponse = input("Voulez-vous continuer? (oui/non)")
while reponse != 'non' :
   reponse = input("Voulez-vous continuer? (oui/non)")
```

7.4 Boucles imbriquées

Tout comme les tests imbriqués sont possibles en Python, les boucles imbriquées le sont également.

Codons un script pour générer les tables de multiplication de 1 à 10 :

```
for i in range(1, 11) :
   for j in range(1, 11) :
      print(i, 'x', j, '=', j *i)
   print()    # un saut de ligne entre chaque table pour la
              # lisibilité
```

7.5 Pour aller plus loin

Python, comme d'autres langages de programmation, a ajouté des fonctionnalités aux structures itératives pour en avoir une gestion plus fine.

7.5.1 Break

Le mot-clé `break` est utilisé dans un `for` ou un `while` dans l'objectif d'interrompre la boucle courante.

```
for i in range(10) :
    print(i)
    break
```

Le script précédent affiche la valeur 0 et s'arrêtera au `break`. Il est de convention d'utiliser un `break` dans un `if` pour arrêter les itérations avec la validité d'une condition.

```
for i in range(10) :
    if i == 5 :
        break
    print(i)
```

Le script précédent affiche les valeurs 0 à 4 et interrompt les itérations dès que la variable `i` prend comme valeur 5.

Remarque

Le break est à utiliser que si ne pouvez pas faire autrement, il ne s'agit pas d'une bonne technique de programmation.

7.5.2 Continue

Le `continue` est un **court-circuit** : il permet de sauter l'itération courante et de passer à la suivante automatiquement sans exécuter les instructions du bloc de la boucle. Il est toujours utilisé dans une structure conditionnelle.

```
for i in range(10) :
    if i == 5 :
        continue
    print(i)
```

Le script précédent affiche les valeurs 0 à 4 puis 6 à 10. Lorsque le compteur arrive à 5, le programme exécute donc les instructions du `if`, c'est-à-dire le `continue`, sautant ainsi cette itération.

Remarque

Le `continue`, comme le `break`, est à utiliser que si ne pouvez pas faire autrement, il ne s'agit pas d'une bonne technique de programmation.

7.5.3 Boucle-else

Python permet de gérer plus finement les boucles interrompues par un `break` avec les "boucle-else". Le principe est assez simple : la boucle est suivie d' un `else`. Les instructions du bloc `else` sont exécutées après la boucle uniquement si cette dernière n'a pas exécuté de `break`, c'est-à-dire qu'elle s'est déroulée normalement. Ce `else` peut s'appliquer autant au `for` qu'au `while`.

```
for i in range(10) :
    if i == 5 :
        break
    print(i)
else :
    print("je rentre dans le else du for") # pas exécuté car le
                                           # for est interrompu
while i < 10 :
    print(i)
    i += 1
else :
    print("je rentre dans le else du while") # exécuté car le
                                             # while n'est pas
                                             # interrompu
```

Normalement, vous n'utilisez que rarement ce type de boucle. Cependant, vous pourrez facilement en lire sur des exemples de scripts ou de la documentation, c'est pourquoi nous vous les avons expliqués.

8. Exercices

8.1 Exercice 1

Écrivez un algorithme qui affiche les vingt premiers termes de la table de multiplication en signalant les multiples de 3 avec un astérisque. Codez le script Python correspondant.

8.2 Exercice 2

Écrivez un algorithme qui calcule la multiplication de deux entiers entrés par l'utilisateur sans utiliser l'opérateur x (i.e. avec des additions successives). Codez le script Python correspondant.

8.3 Exercice 3

Écrivez un algorithme qui récupère plusieurs fois une chaîne de caractères entrée par l'utilisateur et en affiche sa longueur jusqu'à ce que cette entrée corresponde au mot "end". Codez le script Python correspondant.

8.4 Exercice 4

Écrivez un algorithme qui affiche les n premiers carrés, n étant un entier entré par l'utilisateur. Codez le script Python correspondant.

8.5 Exercice 5

Écrivez un algorithme qui implémente le jeu du FizzBuzz : afficher les cents premiers entiers en remplaçant les multiples de trois par Fizz, les multiples de cinq par Buzz et ceux de quinze par FizzBuzz. Codez le script Python correspondant en ajoutant que les sauts de ligne sont uniquement après les multiples de dix.

8.6 Exercice 6

Écrivez un algorithme qui détermine si une chaîne est un palindrome, c'est-à-dire un mot qui se lit aussi bien de gauche à droite que de droite à gauche. Codez le script Python correspondant.

8.7 Exercice 7

Écrivez un algorithme qui affiche un triangle rectangle d'étoiles (triangle avec un angle droit) dont le nombre d'étages est entré par l'utilisateur. Codez le script Python correspondant.

8.8 Exercice 8

Écrivez un algorithme qui affiche un triangle isocèle d'étoiles (triangle avec deux côtés de même longueur) dont le nombre d'étages est entré par l'utilisateur. Codez le script Python correspondant.

Chapitre 5
Les tableaux et structures

1. Introduction

Jusqu'à présent, nous nous sommes préoccupés de variables qui n'avaient aucun lien logique entre elles. Dans ce chapitre, nous allons étudier comment stocker et manipuler plusieurs valeurs qui sont liées entre elles afin de pousser encore plus loin notre réflexion et surtout nos algorithmes et programmes.

2. Les tableaux

Imaginez une liste de courses simple : vous devez acheter douze œufs, du beurre demi-sel, une salade, deux barquettes de fraises et trois tablettes de chocolat. Chaque élément de cette liste est un produit à acheter.

Avec ce que nous avons appris, votre premier instinct serait de créer une variable pour chaque produit :

```
PROGRAMME Liste_course_var
VAR
   produit1 : CHAINE
   produit2 : CHAINE
   produit3 : CHAINE
   produit4 : CHAINE
   produit5 : CHAINE
   i : ENTIER
```

```
DEBUT
      ECRIRE("Entrer un produit à acheter)
      produit1 <- LIRE()
      ECRIRE("Entrer un produit à acheter)
      produit2 <- LIRE()
   ...
FIN
```

Cet algorithme paraît lourd, rébarbatif, peu judicieux... Si vous devez ajouter un produit à acheter, vous allez devoir réécrire cet algorithme en ajoutant une variable `produit6` et la valeur finale du compteur de la structure itérative `POUR`. Vous allez en fait réécrire cet algorithme à chaque fois que vous voulez ajouter ou enlever un produit.

Cette logique va à l'inverse de l'algorithmie : l'algorithme doit prendre en compte le plus de cas possibles pour être le plus générique possible.

Pour pallier ce problème, il existe une structure de données particulière qui permet de stocker plusieurs valeurs de même type : les tableaux. Avec cette nouvelle structure, vous allez également simplifier la saisie et la manipulation de ces valeurs tout en écrivant un algorithme plus compréhensible et donc maintenable.

2.1 Tableaux à une dimension

Une variable de type tableau peut contenir plusieurs valeurs, la seule obligation est que ces valeurs soient de même type en algorithmie. Les tableaux se déclarent en même temps que les autres variables. Un tableau à une dimension représente une liste de valeurs. Un tableau à deux **dimensions** peut être vu comme la feuille d'un tableur avec un nombre de **colonnes** par ligne (chaque ligne a toujours le même nombre de colonnes) en plus du nombre de **lignes** (toujours la première dimension par convention en informatique).

Les dimensions sont des **valeurs entières** qui vont indiquer le nombre de lignes et/ou de cases de la dimension.

Nous commencerons par les tableaux à une dimension, ce que nous appelons, être humain, une liste.

Une variable de type TABLEAU doit être déclarée avec ses dimensions, (chaque dimension étant fixée dans la déclaration), et le type des valeurs qu'il va stocker.

```
VAR
   tab : TABLEAU[1…dim] : type
```

Nous avons déclaré précédemment une variable tab de type TABLEAU qui possède une dimension qui commence à 1 et finit à la valeur dim, donc de dim cases. Chaque valeur de tab sera du type "type".

Pour accéder à une valeur du tableau, que ce soit en lecture ou en écriture, il existe l'opérateur **d'indexation** : [ ]. Le numéro de la case du tableau est appelé son **index**, qui peut être une variable ou une valeur du type entier.

Pour accéder au premier élément d'un tableau, nous utilisons l'opérateur [1] sur le tableau. De manière plus générale, pour accéder au ième éléments d'un tableau tab, il faut utiliser tab[i]. Avec cet opérateur, chaque élément du tableau se comporte comme une variable normale.

Remarque

En algorithmie, le premier indice a comme valeur 1 : nous commençons à compter à partir de 1, ce qui est différent dans la programmation, comme nous l'avons déjà remarqué.

Vous pouvez, comme pour les variables, initialiser un tableau lors de sa déclaration grâce à une paire d'**accolades**.

```
VAR
   jours <- {"lundi", "mardi", "mercredi", "jeudi", "vendredi",
"samedi", "dimanche"} : TABLEAU[1...7] : CHAINE
```

Le tableau jours de l'algorithme précédent représente les jours de la semaine : jours[5] est la valeur "vendredi".

Améliorons maintenant notre liste de courses avec un tableau :

```
PROGRAMME Liste_courses_tableau
VAR
   produits : TABLEAU[1...5]: CHAINE
   i : ENTIER
DEBUT
   POUR i ALLANT de 1 A 5 AU PAS DE 1
```

```
    FAIRE
        ECRIRE("Entrer le produit à acheter")
        produits [i] <- LIRE()
    FINPOUR
FIN
```

Avec cet algorithme, l'utilisateur va entrer les produits les uns après les autres :

12 œufs

1 salade

1 beurre demi-sel

2 barquettes de fraises

3 tablettes de chocolat

Il est dommage qu'il y ait deux informations distinctes dans chaque case : le nom du produit et la quantité.

Les tableaux à deux dimensions vont nous permettre de mieux entrer notre liste de courses en séparent le nom de la quantité, tout en gardant le lien sémantique entre les deux (le 12 sera pour œuf, et non pour salade par exemple).

2.2 Tableaux à deux dimensions

Les tableaux à deux dimensions peuvent être assimilés à un fichier d'un tableur comme Excel par exemple: il s'agit d'un ensemble de **lignes** et chaque ligne possède un nombre de **colonnes**. Les tableaux à deux dimensions sont généralement appelés des **matrices** en informatique.

```
VAR
    matrice : TABLEAU[1..dim1][1...dim2] : type
```

Nous avons déclaré précédemment une variable matrice de type `TABLEAU` qui possède deux dimensions : `dim1` lignes qui ont chacune `dim2` colonnes.

Améliorons de nouveau notre liste de courses :

```
PROGRAMME Liste_courses_tableau_deux_dimensions
VAR
   produits : TABLEAU[1...5][1...2]: CHAINE
   i, j : ENTIER
DEBUT
   POUR i ALLANT de 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrer le produit à acheter")
      produits [i][1] <- LIRE()
      ECRIRE("Entrer la quantité de ce produit à acheter")
      produits [i][2] <- LIRE()
   FINPOUR
   ECRIRE("Voici votre liste de courses")
   POUR i ALLANT de 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE(produits [i][1], " : ", produits [i][2])
   FINPOUR

FIN
```

Votre liste de courses informatisée est donc bien plus élégante maintenant :

Œufs	12
Salade	1
Beurre demi-sel	1
Barquettes de fraises	2
Tablettes de chocolat	3

Cependant, il reste un problème de conception dû aux limites de l'algorithmie : la colonne représentant la quantité doit être une chaîne de caractères car toutes les valeurs des cases du tableau, quelle que soit la ligne ou la colonne, doivent être du **même type**. Or, une quantité devrait être naturellement de type numérique. Nous verrons plus tard dans ce chapitre comment utiliser une autre structure pour résoudre cette incohérence.

2.3 Tableaux à n dimensions

Vous pouvez créer des tableaux avec **plus de deux dimensions**, il n'existe aucune limite, sauf celle de la mémoire de la machine. Il suffit d'ajouter une dimension avec la paire de crochets lors de la déclaration du tableau. Cependant, comprenez bien que plus votre tableau aura de dimensions, plus sa manipulation sera complexe.

Par exemple, nous souhaitons construire un tableau afin de mémoriser les superficies de chaque pièce des appartements d'un pâté d'immeuble. Il y a 9 immeubles, 6 étages par immeuble, 3 appartements par étage et 3 pièces par appartement.

```
PROGRAMME immeuble
VAR
   i, j, k, l  : ENTIER
   superficies : TABLEAU[1...9][1...6][1...3][1...3] : ENTIER
DEBUT
   mat[7][6][1][2] <- 40
   POUR i ALLANT de 1 à 9 AU PAS DE 1 FAIRE
      POUR j ALLANT de 1 à 6 AU PAS DE 1 FAIRE
         POUR k ALLANT de 1 à 3 AU PAS DE 1 FAIRE
            POUR l ALLANT de 1 à 3 AU PAS DE 1
            FAIRE
               ECRIRE("Entrez la superficie de l'immeuble ",
i, " étage ", j, " appartement ", k, " pièce ", l)
               superficies [i][j][k][l] <- LIRE()
            FINPOUR
         FINPOUR
      FINPOUR
   FINPOUR
FIN
```

Cet exemple se traduit par un tableau à quatre dimensions, ce qui demande énormément d'attention lors de sa manipulation, comme le montre son initialisation avec des entrées utilisateur dans l'algorithme précédent.

3. Manipulations simples des tableaux

3.1 Tableaux à une dimension

3.1.1 Parcours

Pour parcourir un tableau, nous devons aller de la première case de la ligne à la dernière. Cela indique donc que nous devons déclarer un compteur et, qui dit compteur, dit structures itératives POUR.

```
PROGRAMME Parcours_tableau_une_dimension
VAR
   jours <- {"lundi", "mardi", "mercredi", "jeudi", "vendredi",
 "samedi", "dimanche"} : TABLEAU[1...7] : CHAINE
   i: ENTIER
DEBUT
   POUR i ALLANT DE 1 A 7 AU PAS DE 1
   FAIRE
      ECRIRE("Le jour ",i," est ", jours[i])
   FINPOUR
FIN
```

Un tableau se **parcourt toujours avec une boucle** POUR où le compteur nous permet de récupérer toutes les valeurs du tableau car il est également égal à l'indice courant de la case.

3.1.2 Recherche

La recherche d'une valeur est une opération courante en informatique. Elle se fait avec un parcours de tableau. À chaque case du tableau, nous allons vérifier si la valeur de l'élément courant est égale à la valeur recherchée.

```
PROGRAMME Recherche_tableau_une_dimension
VAR
   jours <- {"lundi", "mardi", "mercredi", "jeudi", "vendredi",
"samedi", "dimanche"} : TABLEAU[1...7] : CHAINE
   jour_a_chercher : CHAINE
   i: ENTIER
   trouve <- FAUX : BOOLEEN
DEBUT
   ECRIRE("Entrer le nom du jour à chercher")
```

```
    jour_a_chercher <- LIRE()
    i ALLANT DE 1 A 7 AU PAS DE 1
    FAIRE
          SI jours[i] = jour_a_chercher
       ALORS
          trouve <- VRAI
       FINSI
    FINPOUR

    SI trouve = VRAI
    ALORS
       ECRIRE("le jour ", jour_a_chercher, " est présent")
    SINON
       ECRIRE("le jour ", jour_a_chercher, " est absent")
    FINSI
FIN
```

La logique pour rechercher dans ce tableau est la suivante : **nous partons de l'hypothèse que la valeur ne se trouve pas dans le tableau** (`trouve` est initialisé à `FAUX` lors de sa déclaration). Puis nous parcourons notre tableau en comparant ses valeurs à celle que nous recherchons. Si la valeur recherchée apparaît dans le tableau, la variable `trouve` prend comme valeur `VRAI` et sinon elle reste à `FAUX`.

Pourquoi ne pas partir de l'hypothèse que la valeur se trouve dans le tableau (initialiser `trouve` à `VRAI`) ? En partant de cette hypothèse, nous devrons passer `trouve` à FAUX si la valeur actuelle n'est pas celle recherchée. Avec cette affectation, nous aurons de grandes chances de déclarer que la valeur n'est pas présente alors qu'elle l'est. En effet, avec cette hypothèse de départ, le test fatidique sera toujours la comparaison de la valeur de la dernière case du tableau.

3.1.3 Réduction

Réduire un tableau revient à calculer une valeur à partir des valeurs présentes dans le tableau. L'exemple le plus facile pour assimiler ce concept est le calcul de la somme des valeurs des cases d'un tableau.

```
PROGRAMME Somme_tableau
VAR
   tab : TABLEAU[1...12] : ENTIER
   i, somme : ENTIER
DEBUT
   // Saisie des valeurs du tableau
   POUR i ALLANT de 1 à 12 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez la valeur ", i, " du tableau")
      tab[i] <- LIRE()
   FINPOUR
   // calcul de la somme
   somme <- tab[1]
   POUR i ALLANT de 2 à 12 AU PAS DE 1
   FAIRE
         somme <- somme + tab[i]
      FINPOUR
   FINPOUR
   ECRIRE("La somme du tableau est : ", somme)
FIN
```

Dans la réduction d'un tableau, il est déconseillé d'initialiser avec une valeur par défaut la variable de la réduction lors de sa déclaration, sauf avec l'élément neutre de l'opération de réduction. Prenons l'exemple de la somme que vous calculez en tant qu'être humain. Quand vous lisez un tableau à sommes, vous commencez par dire que la somme est égale au premier élément, puis vous lui ajoutez le deuxième, puis le troisième... jusqu'au dernier. Ce calcul est naturel pour un être humain et vous ne vous rendez pas forcément compte de ses étapes. La logique est la même pour la machine. Quelle que soit la réduction que vous cherchez à faire (la somme, la moyenne, la plus petite valeur...), cette réduction est toujours initialisée avec la valeur d'une case du tableau, et généralement avec la valeur de la première case du tableau. Cela permet de travailler uniquement avec des valeurs justes, i.e. contenues dans le tableau, et donc ne pas exécuter une réduction fausse due à une mauvaise initialisation.

3.2 Tableaux à n dimensions

3.2.1 Parcours

Pour tout parcours de tableaux à n dimensions, il existe une règle simple à retenir : **une dimension = une structure itérative** POUR. Deux dimensions donnent un parcours avec deux POUR imbriqués, trois dimensions trois POUR imbriqués, etc.

```
PROGRAMME Parcours_tableau_deux_dimensions
VAR
   matrice : TABLEAU[1...12][1...31] : REELS
   i, j : ENTIER
DEBUT
   // Saisie des valeurs de la matrice
   POUR i ALLANT de 1 à 12 AU PAS DE 1
   FAIRE
      POUR j ALLANT de 1 à 31 AU PAS DE 1
      FAIRE
         ECRIRE("Entrez la valeur de la ligne", i, "
et de la case ", j, " du tableau")
         matrice [ i ][ j ] <- LIRE()
      FINPOUR
   FINPOUR
   // Affichage des valeurs de la matrice
   POUR i ALLANT de 1 à 12 AU PAS DE 1
   FAIRE
      POUR j ALLANT de 1 à 31 AU PAS DE 1
      FAIRE
         ECRIRE(matrice [ i ][ j ])
      FINPOUR
   FINPOUR
FIN
```

3.2.2 Recherche

La recherche dans un tableau d'une dimension ou de n dimensions repose sur la même logique que celle pour un tableau à une dimension. Nous partons du principe que la valeur n'est pas dans le tableau avec un **"flag" de type booléen à** FAUX. Nous parcourons le tableau et si nous trouvons la valeur nous passons le flag à VRAI. À la fin du parcours, il ne nous reste plus qu'à tester la valeur du flag pour savoir si la valeur est présente ou non dans le tableau.

```
PROGRAMME Recherche_tableau_deux_dimensions
VAR
   matrice : TABLEAU[1...12][1...31] : REELS
   i, j : ENTIER
   valeur : REEL
      trouve <- FAUX : BOOLEEN
DEBUT
   // Saisie des valeurs de la matrice
POUR i ALLANT de 1 à 12 AU PAS DE 1
   FAIRE
      POUR j ALLANT de 1 à 31 AU PAS DE 1
      FAIRE
         ECRIRE("Entrez la valeur de la ligne", i, " et
de la case ", j, " du tableau")
         matrice [i][j] <- LIRE()
      FINPOUR
   FINPOUR
   ECRIRE("Entrez la valeur réelle à rechercher")
   valeur <- LIRE()
   // Recherche de la valeur dans la matrice
   POUR i ALLANT de 1 à 12 AU PAS DE 1
   FAIRE
      POUR j ALLANT de 1 à 31 AU PAS DE 1
      FAIRE
         SI tab[i][j] = valeur
         ALORS
            trouve <- VRAI
         FINSI
      FINPOUR
   FINPOUR
   SI trouve = VRAI
   ALORS
      ECRIRE("la valeur est présente dans le tableau)
   SINON
      ECRIRE("le tableau ne contient pas la valeur recherchée")
   FINSI
FIN
```

Nous remarquons donc que la réduction de tableaux à deux dimensions repose également sur la **même logique que celle des tableaux à une dimension**. Ces manipulations simples sont la base des traitements avec des tableaux en informatique. Cependant, ce ne sont pas les plus utilisées finalement : avec des tableaux, nous recherchons d'abord à organiser nos données correctement pour les traiter rapidement, c'est à dire en les triant. Cependant, il nous manque encore quelques notions pour étudier comment trier un tableau proprement et rapidement, notions qui seront étudiées au prochain chapitre.

4. Structures et enregistrements

4.1 Structures

Lorsque nous avons modélisé notre liste de courses avec l'algorithme `Liste_courses_tableau_deux_dimensions`, nous nous sommes aperçus d'un problème de type de données. La quantité est représentée par une chaîne de caractères et non un entier, ce qui n'est pas logique car nous ne pouvons pas faire de calcul sur cette quantité, comme la décrémenter de 1 par exemple.

La `STRUCTURE` algorithmique permet justement de corriger ce problème : dans un tel type, nous pouvons **lier plusieurs variables de types différents ou non**. La `STRUCTURE` permet de lier des variables dans une seule et même variable. Ces variables sont appelées les **champs** de la structure.

```
STRUCTURE nom_structure
DEBUT
   champ1 : type1
   champ2 : type2
   ...
FINSTRUCTURE
```

La définition de la structure se déclare **juste avant le** `PROGRAMME`. La déclaration d'une variable de TYPE `structure` se fait comme pour les variables d'autres types. Nous appelons ces variables des **enregistrements**.

```
VA
   mon_enregistrement : nom_structure
```

Nous pouvons donc modéliser notre liste de courses avec la `STRUCTURE` suivante :

```
STRUCTURE produit
DEBUT
   nom : CHAINE
   quantité : ENTIER
FINSTRUCTURE
```

À la différence des autres types de variables, les variables de type `STRUCTURE` ne doivent pas initialiser une valeur unique mais toutes les valeurs de tous leurs champs. Pour y accéder, nous utilisons **l'opérateur "."** qui permet de manipuler les champs comme des variables simples.

Pour saisir un produit, il faut saisir son nom et sa quantité :

```
PROGRAMME saisie_produit_liste_courses
VAR
   mon_produit : produit
DEBUT
   ECRIRE("Entrez le nom du produit")
   mon_produit.nom <- LIRE()
   ECRIRE("Entrez la quantité du produit")
   mon_produit.quantite <- LIRE()
   ECRIRE(mon_produit.nom, " avec une quantité de ",
mon_produit.quantite)
FIN
```

La modélisation des produits de notre liste de courses est maintenant largement plus propre.

4.2 Structures imbriquées

Les champs d'une structure peuvent être de n'importe quel type de donnée, donc pourquoi pas de type STRUCTURE. Dans ce cas, **la structure imbriquée doit être déclarée avant la structure qui la contient** (sinon elle ne sait pas qu'elle existe) et nous devons toujours utiliser l'opérateur "." pour accéder aux champs. Modélisons une personne avec sa date de naissance. Une date étant composée de trois valeurs, nous la représentons donc avec une structure.

```
PROGRAMME Personne_structure
STRUCTURE date
DEBUT
   jour : ENTIER
   mois : ENTIER
   annee : ENTIER
FINSTRUCTURE
STRUCTURE personne
DEBUT
   prenom : CHAINE
   nom : CHAINE
   naissance : Date
FINSTRUCTURE
VAR
   toto : personne
DEBUT
   ECRIRE("Saisie de l'identité de toto")
   ECRIRE("Entrer le nom de toto")
   personne.nom <- LIRE()
   ECRIRE("Entrer le prénom de toto")
   personne.prenom <- LIRE()
   // Trois saisies pour la date de naissance, car trois champs
   ECRIRE("Entrer le jour de naissance de toto")
   toto.naissance.jour <- LIRE()
   ECRIRE("Entrer le mois de naissance de toto")
   toto.naissance.mois <- LIRE()
   ECRIRE("Entrer l'année de  naissance de toto")
   toto.naissance.annee <- LIRE()
   ECRIRE(toto.nom, " ", toto.prenom, " est né le ",
toto.naissance.jour,"/", toto.naissance.mois, "/",
toto.naissance.jour)
FIN
```

Nous remarquons que la manipulation de structures imbriquées requiert **plusieurs fois d'affilé l'utilisation de l'opérateur "."** pour accéder aux champs désirés.

4.3 Structures et tableaux

4.3.1 Structure contenant un tableau

Comme nous l'avons vu dans la section précédente, les champs d'une structure peuvent être de n'importe quel type de donnée, donc pourquoi pas d'un type tableau. Modélisons la structure représentant un livre qui possède un titre et plusieurs éditions. Les éditions sont bien entendu un tableau, vu qu'elles sont multiples et de même type.

```
PROGRAMME livre_strucutre
STRUCTURE livre
DEBUT
   titre : CHAINE
   editions : TABLEAU[1...5] : CHAINE
FINSTRUCTURE
VAR
   mon_livre : livre
   i : ENTIER
DEBUT
   ECRIRE("Saisie de votre livre")
   ECRIRE("Entrer le titre du livre")
   mon_livre.titre <- LIRE()
   POUR i ALLANT DE 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrer l'édition numéro ", i)
      mon_livre.editions[i] <- LIRE()
   FINPOUR
FIN
```

Nous pouvons voir que la logique de manipulation des tableaux reste la même, que les tableaux soient des variables ou des champs.

4.3.2 Tableau de structures

Vous pouvez également créer des tableaux qui ont des valeurs de type STRUCTURE. Reprenons la saisie correcte de notre liste de courses pour illustrer ce principe.

```
PROGRAMME saisie_liste_courses_structure
VAR
   produits : TABLEAU[1…5]: produit
   i : ENTIER
DEBUT
   POUR i ALLANT de 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez le nom du produit")
      produits [i].nom <- LIRE()
      ECRIRE("Entrez la quantité du produit")
      produits [i].quantite <- LIRE()
   FINPOUR
FIN
```

Notre liste de courses est maintenant propre en informatique !

Remarque

Le langage Python n'implémente pas cette structure, il utilise pour la remplacer la programmation orientée objet dont nous ferons une introduction au chapitre Commencer avec l'objet de cet ouvrage.

5. Mettons en pratique avec Python

5.1 Tableau = liste

Les tableaux en algorithmie sont appelés **listes** en Python.

Une liste est une variable qui contient plusieurs expressions (ou valeurs) qui ne sont **pas forcément du même type**, contrairement à l'algorithmie. Ce principe respecte donc le **typage dynamique** du langage.

Une liste est un type de donnée dit **mutable** : elle peut changer de valeur et de taille.

Une liste a également un autre avantage sur le tableau : elle est de **taille dynamique**, nul besoin d'en déclarer la taille ! La gestion des indices se fait également automatiquement.

Remarque

Dans tous les langages de programmation, hors les implémentations du langages SQL comme par exemple TransactSQL, nous commençons à compter ***à partir de 0****. Le premier élément d'une liste est donc d'indice 0 (à la position 0 de la liste) en Python.*

Les indices en Python peuvent être positifs ou négatifs :

- **Positifs** pour aller du premier élément d'une liste aux autres.
- **Négatifs** pour commencer par le dernier élément de la liste.

Ainsi, si nous voulons accéder au **dernier élément** d'une liste, nous demanderons l'élément numéro "**taille de la liste -1**" en indice positif ou tout simplement l'élément d'indice **-1** pour les indices négatifs.

Une liste se déclare avec l'opérateur **d'indexation [ ]**. Si la liste est vide, nous ne mettons rien entre les crochets, sinon on y ajoute la liste des expressions séparées par des virgules. L'accès aux éléments d'une liste se fait comme en algorithmie grâce à l'opérateur d'indexation.

```
ma_liste_vide = []
ma_liste_taille_1 = ["je suis le premier élément d'indice 0"]
ma_liste_taille_n = [1, "deux", True, 3.14]
print("Deuxième élément de la liste de taille n :",
ma_liste_taille_n[1])#affiche Deuxième élément de la liste
de taille n : deux
```

Comme en algorithmie, Python donne également la possibilité de créer des tableaux à n dimensions, une paire de crochets par dimension.

```
board = []
# Initialisation
for line in range(3) :
    board.append([])
    for col in range(3) :
        board[line].append(0)
# Affichage
for line in board :
```

```
    for col in line :
        print(col, end=" ")
    print()
```

Le script précédent initialise un plateau de jeu de 3x3 cases initialisées à 0. Notons que pour créer notre liste à deux dimensions, nous créons **une liste vite au départ pour modéliser les lignes**. Ensuite, nous ajoutons **une liste comme élément de cette liste pour chaque ligne pour modéliser les colonnes**. Puis nous terminons par insérer les valeurs de chaque colonne de chaque ligne.

5.1.1 Parcours

En Python, il existe trois méthodes principales pour parcourir une liste :

- `for in` : parcours de chaque valeur de la liste.
- `for range` : parcours de chaque indice de la liste.
- `for enumerate` : permet de récupérer à chaque tour l'élément et son indice.

```
ma_liste = [-2, 3, 11]
for x in ma_liste :
    print(x, end=" ")    # -2 3 11
print()                  # simple saut de ligne pour l'affichage
for i,x in enumerate(ma_liste) :
    print("Element",i,":",  end=" ")    # Element 0 : -2
Element 1 : 3 Element 2 : 11
print()
for i in range(3):     # 3 étant la taille de liste
    print(ma_liste[i], end=" ")    # -2 3 11
print()
```

Python permet également de **parcourir deux listes en même temps**, dans une même structure itérative, avec la fonction `zip`. La boucle `for` s'arrête au dernier élément de la liste ayant la plus petite taille.

```
ma_liste = [-2, 3, 11]
mon_autre_liste = ['a','b']

for x, y in zip(ma_liste, mon_autre_liste) :
    print(x,"-", y, end=" ")    # -2 - a 3 -b
print()
```

5.1.2 Opérations sur les listes

Python possède une multiple d'opérateurs déjà implémentés pour manipuler les listes dont voici les principaux :

Opérateur	Résultat
`x in l`	Teste si la valeur de x est dans la liste l.
`x not in l`	Teste si la valeur de x n'est pas dans la liste l.
`l1 + l2` ou `l1.extend(l2)`	Concaténation de la liste l1 et l2.
`l * n`	Concaténation de n copies de l.
`len(l)`	Longueur ou taille de la liste l.
`min(l)`	Plus petit élément de l.
`max(l)`	Plus grand élément de l.
`l.count(x)`	Nombre d'occurrences de la valeur de x dans l.
`l.index(x)`	Premier indice de la position de la valeur de x dans la liste l.
`l.append(x)`	Ajout de x à la fin de la liste l.
`l.insert(i, x)`	Insertion à l'indice de valeur i la valeur de l'élément x dans la liste l.
`l.clear()`	l devient une liste vide, sans élément.
`l.remove(x)`	Suppression de la valeur de x dans la liste l.
`l.pop(i)`	Renvoie la valeur de l'élément d'indice i et la supprime de la liste.
`l.reverse()`	Inversion de la liste l.
`l.sort()`	Trie la liste l par ordre ascendant.

Remarque

Les opérations min et max fonctionnement correctement quel que soit le type des éléments de la liste car tout type est transposable en type numérique. Python peut donc comparer des entiers et des chaînes de caractères par exemple.

```
ma_liste = [-2, 3, 11]
print(ma_liste)   # [-2, 3 ,11]
print("Taille de ma liste", len(ma_liste))   # 3
print("Minimum de ma liste", min(ma_liste))   # -2
print("Indice de l'élément de la valeur 11",
ma_liste.index(11))        # 2
ma_liste.append(1)
print(ma_liste)   # [-2, 3 ,11, 1]
ma_liste.insert(1, 42)
print(ma_liste)   # [-2, 42, 3 ,11, 1]
ma_liste.sort()
print(ma_liste)   # [-2, 1, 3 , 11, 42]
ma_liste.clear()
print(ma_liste)   # []
```

5.1.3 Copie

En Python, lorsque nous affectons une liste avec une autre, les deux listes sont liées : si nous changeons l'une des deux listes, nous changeons automatiquement l'autre liste.

```
ma_liste = [-2, 3, 11]
ma_copie = ma_list
ma_liste.append(1)
print(ma_liste, "vs", ma_copie)   # [-2, 3, 11, 1] vs [-2, 3, 11, 1]
ma_copie.clear()
print(ma_liste, "vs", ma_copie)   # [] vs []
```

Pour casser ce lien entre ces deux listes, il existe trois solutions :

- Utiliser la fonction `copy`.
- Utiliser la fonction `list`.
- Utiliser la fonction `extend` de la liste.

```
ma_liste = [-2, 3, 11]
# fonction copy
ma_copie_separee = ma_liste.copy()
# fonction list
ma_deuxieme_copie = list(ma_liste)
# Etendre une liste vide avec la liste à copier
ma_troisieme_copie = []
ma_troisieme_copie.extend(ma_liste)
ma_liste.append(1)
```

```
print(ma_liste, "vs", ma_copie_separee)    # [-2, 3, 11, 1] vs
[-2, 3, 11]
print(ma_liste, "vs", ma_deuxieme_copie)    # [-2, 3, 11, 1] vs
[-2, 3, 11]
print(ma_liste, "vs", ma_troisieme_copie)    # [-2, 3, 11, 1] vs
[-2, 3, 11]
```

Remarque

En Python, l'affection d'une liste ne crée pas un nouveau pointeur, les deux listes seront en fait le même pointeur. Le chapitre Les fichiers détaillera cette notion de pointeurs en détail pour mieux la comprendre.

5.1.4 Pour aller plus loin : l'intention

Python intègre une manière très élégante de créer des listes avec des valeurs qui peuvent être calculées grâce à une boucle : les **listes en intention**.

```
[valeur for x in sequence if condition]
```

La liste en intention est constituée des valeurs `valeur` pour tous les éléments `x` de la boucle `for` qui valident la condition du `if`.

Imaginons que vous vouliez créer une liste des dix premiers entiers positifs ou nuls. Vous écririez le code suivant :

```
l = []
for i in range(10) :
    l.append(i)
```

Nous pouvons, grâce à la syntaxe des listes en intention de Python, écrire ce script en une ligne :

```
l = [i for i in range(10)]
print(l)        # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si nous ne voulons récupérer que les entiers pairs, il suffit d'ajouter la condition :

```
l_pair = [i for i in range(10) if i % 2 == 0]
print(l_pair)       # [0, 2, 4, 6, 8]
```

Et pour finir, la liste des carrés des cinq premiers entiers pairs :

```
l_carre_pair = [i**2 for i in range(10) if i % 2 == 0]
print(l_carre_pair)      # [0, 4, 16, 36, 64]
```

Souvenez-vous de la création du plateau de jeu vue précédemment. Les listes en intention démontrent toute leur élégance avec cet exemple :

```
board = [[0]*3 for i in range(3)]
```

Notre plateau est composé de trois colonnes à 0 pour chacune des trois lignes et ce en une ligne de code.

5.2 Tuple

Lors du chapitre sur les structures conditionnelles, vous avez lu la remarque de limiter l'usage des parenthèses au maximum dans les expressions (hors calculs mathématiques le nécessitant bien sûr). En effet, les parenthèses en Python sont utilisées pour la déclaration des **tuples** : un type de liste **immuable** (non modifiable).

Un tuple doit être initialisé lors de sa déclaration car nous ne pouvons pas en changer ni la taille ni la valeur de ses éléments.

```
mon_tuple = (1, 'a', 8)
print(mon_tuple[1])   # a
print(len(mon_tuple))   # 3
mon_tuple.append(5)   # erreur arrêt du script
```

Un tuple peut être manipulé avec toutes les opérations des listes qui n'en changent ni la taille, ni la valeur d'un ou de plusieurs de ses éléments.

5.3 Slicing

5.3.1 Listes et tuples

L'opérateur d'indexation en Python est très puissant : il ne sert pas qu'à accéder à un élément d'une liste, il permet également de créer des listes à partir d'une autre, de supprimer plusieurs éléments ou de les remplacer. Ces manipulations sont appelées le **slicing** en Python.

Opérateur	Résultat
`l[i]`	Valeur de l'élément i de la liste l
`l[i] = x`	Remplacement de la valeur de l'élément i de la liste l par la valeur de x
`l[i:j]`	Liste contenant les valeurs des éléments d'indice i jusqu'à l'indice j exclus de la liste l
`l[i:j] = l2`	Remplacement des éléments d'indice i jusqu'à l'indice j exclus de la liste l par ceux de la liste l2
`l[i:j:k]`	Liste contenant les valeurs des éléments d'indice i jusqu'à l'indice j exclus au pas de k de la liste l
`del(l[i])`	Suppression de l'élément d'indice i de la liste l
`del(l[i:j])`	Suppression des éléments d'indice i jusqu'à l'indice j exclus de la liste l

Notons que les indices i, j et k peuvent aussi bien être positifs que négatifs en Python. Un exemple qui peut paraître complexe devient extrêmement simple : inverser une liste sans l'opération reverse. Le slicing répond à ce défi en une ligne :

```
ma_liste = ma_liste[-1::-1]
```

Nous partons du dernier élément jusqu'au premier au pas de -1 pour retourner la liste.

Les indices i, j et k sont **intelligemment facultatifs** : nous n'avons pas l'obligation de tous les déclarer. Par défaut, i vaut 0, c'est-à-dire le premier indice, j la valeur du dernier indice et le pas k vaut 1, ce qui fait que nous devons les déclarer uniquement quand nous ne les voulons pas à ces valeurs.

```
ma_liste = [-2, 3, 11, 2, 0, 11]
print(ma_liste[-1::-1])      # [11, 0, 2, 11, 3, -2]
print(ma_liste[1::])         # [3, 11, 2, 0, 11]
print(ma_liste[1::2])        # [3, 2, 11]
print(ma_liste[::2])         # [-2, 11, 0]
```

5.3.2 Retour sur les chaînes

En Python, les chaînes sont considérées comme des listes composées uniquement de caractères.

De ce fait, les **opérations sur les listes et le slicing sont aussi disponibles** pour les chaînes.

```
str1 = "toto"
lg= len(str1)
print(lg)              # 4
str1 += 's'
print(str1)            # totos
str1 *= 4
print(str1)            # totostotostotostotos
print(str1[0], str1[-1])     # t s
print(str1[0:3], str1[0:6:2])   # tot tts
```

5.4 Dictionnaire

Le **dictionnaire** en Python est un tableau associatif **clé/valeur**. Pour accéder à une valeur d'un dictionnaire, nous ne passons pas par son indice mais par sa **clé**. Les clés et les valeurs sont définies par le développeur.

Voyez le dictionnaire comme un répertoire téléphonique. Lorsque vous cherchez le numéro d'une personne, vous utilisez son nom pour le trouver. Le nom est la clé et le numéro la valeur.

5.4.1 Déclaration et accès

Pour déclarer un dictionnaire, nous utilisons les accolades `{}`.

```
mon_dico = {}   # dictionnaire vide
```

L'accès aux valeurs se fait également avec l'opérateur d'indexation `[]`.

Prenons un exemple de dictionnaire permettant de lier des traductions de termes anglais en français.

```
mon_dico = {}
mon_dico['computer'] = 'ordinateur'
mon_dico['mouse'] = 'souris'
mon_dico['keyboard'] = 'clavier'
print(mon_dico['computer'])      # ordinateur
```

5.4.2 Opérations

Comme pour les listes, Python implémente une multitude d'opérateurs utiles pour manipuler les dictionnaires dont voici les principales :

Opérateur	Résultat
`x in dico`	Teste si la clé x est dans le dictionnaire dico
`x not in dico`	Teste si la clé x n'est pas dans le dictionnaire dico
`len(dico)`	Longueur ou taille du dictionnaire dico
`dico[x]`	Retourne la valeur de la clé x. Si n'est pas une clé, alors retourne une erreur
`del dico[x]`	Supprime la valeur de la clé x et la clé. Si n'est pas une clé, alors retourne une erreur
`dico.clear()`	Vide le dictionnaire dico
`dico.keys()`	Retourne la liste des clés du dictionnaire dico
`dico.values()`	Retourne la liste des valeurs, sans le lien avec les clés, du dictionnaire dico
`dico.items()`	Retourne le couple clé-valeur de chaque élément du dictionnaire

Agrémentons notre traduction anglais-français en faisant participer l'utilisateur.

```
mon_dico = {}
mon_dico['computer'] = 'ordinateur'
mon_dico['mouse'] = 'souris'
mon_dico['keyboard'] = 'clavier'
print(mon_dico['computer'])      # ordinateur
print("Entrez le mot en anglais")
eng = input()
if eng not in mon_dico :
   mon_dico[eng] = input("Entrez sa traduction en français")
else :
   print("Le mot", eng," a déjà une traduction")
```

Afin de ne pas effacer une traduction, nous vérifions avant que le mot en anglais n'existe pas déjà en tant que clé.

5.4.3 Parcours

Le parcours d'un dictionnaire se fait avec une boucle `for` comme pour les listes, mais nous itérons sur les clés et non les indices.

Pour ce faire, nous pouvons utiliser trois méthodes :

- Parcourir les clés avec `for in`.
- Parcourir les clés avec la fonction `keys()`.
- Parcourir les clés et les valeurs avec la fonction `items()`.

```
mon_dico = {}
mon_dico['computer'] = 'ordinateur'
mon_dico['mouse'] = 'souris'
mon_dico['keyboard'] = 'clavier'
# clé simple
for key in mon_dico :
   print(key, ":", mon_dico[key])
# avec l'opération keys()
for key in mon_dico.keys() :
   print(key, ":", mon_dico[key])
#avec l'opération items()
for key, value in mon_dico.items() :
   print(key, ":", value)
```

5.4.4 Pour aller plus loin : l'intention

Tout comme les listes, les dictionnaires peuvent être créés en intention :

```
{clex, y : valeur x, y for x, y in sequence if condtion}
```

Prenons comme exemple un dictionnaire contenant les lettres en majuscules en clé et leur indice de création en valeur.

```
lettres = {chr(i+64):i for i in range(1,6)}
print(lettres)      # {'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5}
```

6. Exercices

6.1 Exercice 1

Écrivez un algorithme qui recherche dans un tableau d'entiers la plus grande et la plus petite valeur du tableau. Codez le script Python correspondant.

6.2 Exercice 2

Écrivez un algorithme qui recherche dans un tableau d'entiers les indices de la plus grande et de la plus petite valeur du tableau. Codez le script Python correspondant.

6.3 Exercice 3

Écrivez un algorithme qui calcule la moyenne des valeurs d'un tableau d'entiers. Codez le script Python correspondant.

6.4 Exercice 4

Écrivez un algorithme qui calcule le nombre d'occurrences d'une valeur entrée par l'utilisateur dans un tableau d'entiers (le nombre de fois que la valeur apparaît dans le tableau). Codez le script Python correspondant.

6.5 Exercice 5

Écrivez un algorithme qui réalise l'inversion des éléments d'un tableau sans utiliser de tableau intermédiaire. Codez le script Python correspondant.

6.6 Exercice 6

Écrivez un algorithme qui permet de représenter un niveau d'un jeu. Chaque niveau est composé d'un plateau de jeu (une matrice de dix par dix caractères), d'un nombre d'objets et d'un boss. Un boss possède comme information un nom et des points de vie.

6.7 Exercice 7

Le drapeau hollandais : le tableau de cet algorithme ne contient que les valeurs entières 1, 2 et 3. Vous devez réarranger les valeurs de ce tableau afin que le 1 soit en premier, suivi des 2 et pour finir les 3. Par exemple, le tableau suivant {3,2,1,2,3,1,1,3} devient {1,1,1,2,2,3,3,3}. Écrivez en premier l'algorithme du drapeau hollandais puis codez le script Python correspondant.

6.8 Exercice 8

Écrivez un algorithme qui calcule la somme de deux matrices d'entiers de même taille : la matrice résultat aura la même taille que les deux matrices et contiendra l'addition des deux éléments des matrices dans ses cases. Codez le script Python correspondant.

6.9 Exercice 9

Écrivez un algorithme qui calcule la somme de chaque colonne d'une matrice d'entiers dans un tableau à une dimension. Codez le script Python correspondant.

6.10 Exercice 10

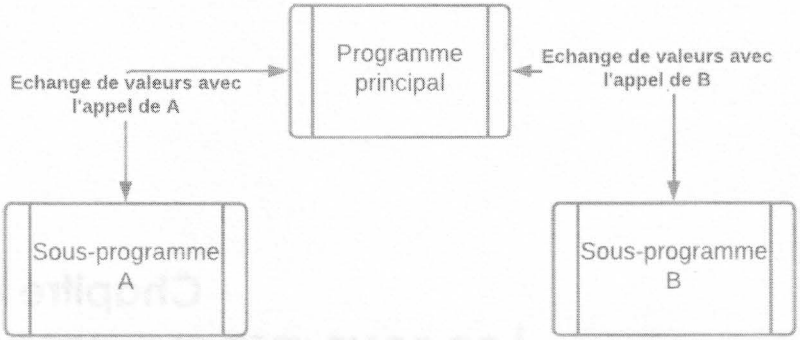
Écrivez un algorithme qui calcule la somme de chaque ligne d'une matrice d'entiers dans un tableau à une dimension. Codez le script Python correspondant.

Chapitre 6
Les sous-programmes

1. Procédures et fonctions

Vous êtes encore en train d'apprendre à modéliser votre pensée en informatique avec des problèmes assez simples et dans ce chapitre vous allez aller un plus loin dans votre réflexion. Imaginez un programme comme un logiciel de vidéo à la demande ou encore un client de messagerie ? Ne pensez-vous pas que ces programmes contiennent des millions, voire des milliards d'instructions ? Comment donc se retrouver facilement dans cette nuée d'instructions ?

Une première réponse est assez simple : découpez votre logique en petites parties qui seront appelées à la demande. Cette découpe permet de fortement réduire la complexité du programme en se focalisant uniquement sur des sous-problèmes plus simples à résoudre. Ces parties sont les **sous-programmes** qui seront appelés dans le **programme principal**, le point d'accès de l'application, comme le représente la figure suivante.



Programme principal et sous-programmes

En algorithmie, les sous-programmes se déclarent avant le programme comme suit :

```
SOUS-PROGRAMME sous_prog_A
VAR
   ...
DEBUT
   ...
FIN
PROGRAMME principal
VAR
   ...
DEBUT
   ...
   sous_prog_A
FIN
```

En réalité, un programme n'est que rarement composé d'une seule série d'instructions qui s'enchaînent les unes après les autres. Pour pouvoir créer et maintenir correctement une application, les développeurs décomposent toujours le programme principal en plusieurs sous-programmes qui vont être appelés par ce dernier. Les sous-programmes représentent en fait une petite fonctionnalité du programme. Ils peuvent également s'appeler entre eux pour définir une hiérarchie de tâches.

Prenons comme exemple un jeu simple : le jeu du pendu. Décomposons-le en termes de fonctionnalités :

- Saisie du mot à deviner.
- Initialisation du mot masqué.
- Lancement de la partie :
  - Affichage du mot masqué.
  - Saisie de la lettre de l'utilisateur.
  - Traitement du mot masqué et du nombre d'essais.
  - Test pour déterminer si le mot est trouvé ou non.

Nous voyons bien, avec cette découpe, que nous n'abordons pas d'un seul coup tout le jeu du pendu, nous le construisons pas à pas, petite fonctionnalité par fonctionnalité. Cela rend donc notre développement plus aisé.

Pour résumer, les sous-programmes ont deux avantages indéniables :

- L'organisation du code est **simplifiée** : un sous-programme a une fonctionnalité bien précise, ce qui permet donc de retrouver facilement les instructions de cette fonctionnalité au besoin.
- Un sous-programme peut être **appelé plusieurs fois** : les copier/coller d'instructions sont donc inutiles ce qui permet donc un programme plus efficient et largement plus maintenable.

Il existe deux types de sous-programmes : les **procédures** et les **fonctions**. Allons les découvrir plus en détail.

1.1 Les procédures

Les procédures sont le premier type de sous-programme. Une procédure permet d'écrire un **ensemble d'instructions et de les exécuter avec une seule instruction**, lorsque nous appelons la procédure.

Une procédure repose aussi sur les principes vus dans cet ouvrage, il s'agit juste d'une organisation différente de notre algorithme et de notre code. Elle peut manipuler des données avec des itérations et/ou des conditionnelles et elle peut échanger des valeurs avec l'algorithme principal. Cette communication se fait via les paramètres.

1.1.1 Paramètres

Les paramètres sont des **variables dédiées à la procédure**. Selon le type de paramètre, la procédure peut récupérer des valeurs de l'algorithme principal et/ou lui en envoyer :

- Les paramètres **d'entrée** : les valeurs que le programme principal envoie à la procédure qui peut les utiliser.
- Les paramètres de **sortie** : les valeurs que la procédure envoie à l'algorithme principal.
- Les paramètres **d'entrée/sortie** : les valeurs initialisées par l'algorithme principal puis modifiées et renvoyées par la procédure.

Du fait de la décomposition d'un programme en sous-programmes avec un système de communication par passage de paramètres, nous devons discuter à nouveau de la portée des variables.

Un sous-programme possède ses propres variables, tout comme le programme principal. Ces variables ne sont connues que du sous-programme, le programme principal ne peut donc pas y accéder. Il en va de même avec les variables du programme principal qui ne sont pas accessibles par les sous-programmes. Ces variables sont dites **locales** au programme ou sous-programme.

Chaque sous-programme est l'équivalent d'une **boîte noire** pour les autres sous-programmes et programmes, ils peuvent l'appeler mais n'ont aucune connaissance ni accès sur ce qu'il contient.

Pour qu'une variable soit utilisable par tous les sous-programmes et le programme, il faut qu'elle soit déclarée comme **globale**, c'est-à-dire en premier dans notre algorithme :

```
VAR...          //déclaration des variables globales
SOUS-PROGRAMME sous_prog_A
VAR
   ...   // déclaration des variables locales à sous_prog_A
DEBUT
   ...
FIN
PROGRAMME principal
VAR
   ...   // déclaration des variables locales à principal
DEBUT
   ...
   sous_prog_A
FIN
```

1.1.2 Déclaration

Une procédure est un algorithme, comme ceux que nous avons vus précédemment, identifié par un nom unique. À la différence des algorithmes, nous définissons également ses paramètres :

```
PROCEDURE nom (E : var1 : type1 , S : var2 : type2 ,
      E/S : var3 : type3)
VAR
   ...      // variables locales à la procédure
DEBUT
   ...      // instructions de la procédure
FIN
```

Commençons par un exemple simple avec deux paramètres d'entrée (en lecture) et deux paramètres de sortie (en écriture : une procédure calculant le périmètre et l'aire d'un rectangle :

```
PROCEDURE rectangle (E : la, lo : ENTIER , S : perimetre, aire : REEL)
DEBUT
   perimetre <- 2 * la + 2 * lo
   aire <- la * lo
FIN
```

Cette procédure prend en entrée la longueur et la largeur d'un rectangle et calcule lors de son appel le périmètre et l'aire dans les variables passées en paramètres de sortie.

Un exemple simple de paramètre d'entrée/sortie est une procédure qui double la valeur d'un entier passé en paramètre. Cette procédure récupère la valeur de l'entier au début et la modifie en la doublant :

```
PROCEDURE double (E/S : n : ENTIER)
DEBUT
   n <- n *2
FIN
```

Continuons maintenant l'exemple du pendu avec l'étape "Affichage du mot masqué" qui peut se traduire par la procédure suivante :

```
PROCEDURE afficher_mot_masque (E : mot : CHAINE)
DEBUT
   ECRIRE("Voici le mot à deviner")
   ECRIRE(mot)
FIN
```

Cette procédure est assez simple, elle prend en paramètre le mot à afficher, qui est donc une entrée car il est en lecture seule, et l'affiche à l'utilisateur.

Augmentons un peu la complexité des procédures de notre pendu. L'étape "Traitement du mot masqué et du nombre d'essais" demande à récupérer la lettre de l'utilisateur et le mot masqué actuel. La lettre est donc une entrée, comme le mot en clair, et le mot initial et le nombre d'essais restent des entrées/sorties car la procédure en récupère les valeurs pour les modifier par la suite, tout en calculant le nombre d'essais qu'il reste à l'utilisateur. Voici la procédure correspondante :

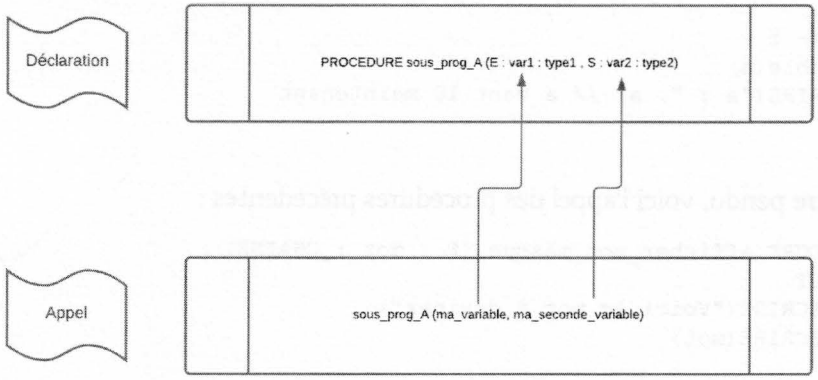
```
PROCEDURE calcul_masquage (E/S : mot : CHAINE, E/S : essai :
ENTIER, E : lettre : CARACTERE, E : mot_clair : CHAINE)
VAR i : ENTIER
tmp : CHAINE
trouve : BOOLEEN
DEBUT
   i <- 1
   tmp <- ""
   trouve <- FAUX
   POUR i ALLANT de 1 à LONGUEUR(mot_clair) AU PAS DE 1
```

```
    FAIRE
       SI EXTRAIRE(mot, i, 1) = lettre
       ALORS
          CONCATENER(tmp, lettre)
          trouve <- VRAI
       SINON
          CONCATENER(tmp, EXTRAIRE(mot,i,1))
       FINSI
    FINPOUR
    SI trouve = FAUX
    ALORS
       essai <- essai -1
    FINSI
    mot <- tmp
FIN
```

Passons maintenant à l'appel de procédure par un programme principal ou un autre sous-programme.

1.1.3 Appel

Les procédures seules ne servent pas à grand-chose, il faut qu'elles soient utilisées pour gagner en intérêt. Une procédure est tout simplement appelée avec une instruction représentant son nom et ses paramètres entre parenthèses dans l'ordre dans lequel ils sont déclarés, comme la montre la figure ci-dessous. Les paramètres peuvent être, lors de l'appel, des variables déclarées ou des valeurs fixes.



Appel d'une procédure

Nous pouvons appeler notre procédure précédente `rectangle`, comme suit :

```
PROCEDURE rectangle (E : la, lo : ENTIER , S : perimetre, aire : REEL)
DEBUT
   perimetre <- 2 * la + 2 * lo
   aire <- la * lo
FIN
PROGRAMME calcul-rectangle
VAR perimetre, aire : REEL
a,b : ENTIER
DEBUT
   rectangle(2, 3, perimetre, aire) // appel avec valeurs
   ECRIRE("périmètre : ", perimetre, " aire : ", aire)

   a <- 5
   b <- 9
   rectangle(a, b, perimetre, aire) // appel avec variables
   ECRIRE("périmètre : ", perimetre, " aire : ", aire)

FIN
```

La procédure `double` qui prend un paramètre en entrée/sortie s'utilise comme suit :

```
PROCEDURE double (E/S : n : ENTIER)
DEBUT
   n <- n *2
FIN
PROGRAMME calcul-double
VAR a : ENTIER
DEBUT
   a <- 5
   double(a)
   ECRIRE("a : ", a) // a vaut 10 maintenant

FIN
```

Pour notre pendu, voici l'appel des procédures précédentes :

```
PROCEDURE afficher_mot_masque (E : mot : CHAINE)
  DEBUT
     ECRIRE("Voici le mot à deviner")
     ECRIRE(mot)
  FIN
PROCEDURE calcul_masquage (E/S : mot : CHAINE, E/S : essai :
ENTIER, E : lettre : CARACTERE, E : mot_clair : CHAINE)
```

```
  VAR i : ENTIER
  tmp : CHAINE
  trouve : BOOLEEN
  DEBUT
     i <- 1
     tmp <- ""
     trouve <- FAUX
     POUR i ALLANT de 1 à LONGUEUR(mot_clair) AU PAS DE 1
FAIRE
              SI EXTRAIRE(mot, i, 1) = lettre
              ALORS
                CONCATENER(tmp, lettre)
                trouve <- VRAI
              SINON
                CONCATENER(tmp, EXTRAIRE(mot,i,1))
              FINSI
           FINPOUR
           SI trouve = FAUX
           ALORS
              essai <- essai -1
           FINSI
mot <- tmp FIN
PROGRAMME pendu
VAR mot, masque : CHAINE
fin_partie : BOOLEEN
essai : ENTIER
DEBUT
   fin_partie <- FAUX
   essai<- 7
   ...
   TANT QUE NON fin_partie
   FAIRE
      ...
      afficher_mot_masque(masque)
      ...
      calcul_masquage(masque, essai, lettre, mot)
   FINTANTQUE
FIN
```

Avec ces appels, nous remarquons que les noms des variables et le nom des paramètres ne sont pas forcément les mêmes. Ici, l'important est la valeur et le type des données envoyées et non l'identifiant car il est local au programme.

1.1.4 Les tableaux en paramètres

Quel que soit le type de sous-programme, les paramètres représentant des tableaux sont des **paramètres spéciaux**. En effet, pour manipuler un tableau, nous avons besoin du tableau et de sa taille. Ainsi, vous devez passer en plus du tableau la **taille** de ce dernier, comme le montre la procédure suivante qui affiche un tableau passé en paramètre.

```
PROCEDURE affiche_tableau(E : tab : TABLEAU [ ] : ENTIER, E : nb : ENTIER)
VAR i : ENTIER
DEBUT
   POUR i ALLANT DE 1 à nb AU PAS DE 1
   FAIRE
      ECRIRE(tab[i])
   FINPOUR
FIN
```

Maintenant que nous avons vu les procédures, étudions le deuxième type de sous-programme, les fonctions.

1.2 Les fonctions

Les fonctions sont le type de sous-programmes le plus utilisé aujourd'hui en informatique. Leur architecture est la même que celle des procédures. Ce sont juste des procédures restreintes :

- Les paramètres sont **uniquement en entrée**.
- Une fonction **renvoie toujours une et une seule valeur** au programme qui l'appelle.

1.2.1 Déclaration

La déclaration d'une fonction est plus simple que celle d'une procédure : les paramètres étant uniquement en entrée, nous n'avons pas besoin de spécifier qu'ils sont entrés avec E. Une fonction se déclare par son nom, ses paramètres et son type de retour :

```
FONCTION ma_fonction(a,b : TYPE, c,d : TYPE) : typeretour
VAR
   ...
```

```
DEBUT
   ...
   RETOURNER(...)
FIN
```

`typeretour` indique le type de la valeur retournée par la fonction. Cette valeur est renvoyée au programme appelant avec l'instruction `RETOURNER()`.

Créons notre première fonction simple qui va renvoyer le plus grand de deux entiers passés en paramètres :

```
FONCTION maximum(a,b : ENTIER) : ENTIER
DEBUT
   SI (a > b)
   ALORS
      RETOURNER(a)
   SINON
      RETOURNER(b)
   FINSI
FIN
```

Cette fonction compare les valeurs des paramètres de type entier `a` et `b` et retourne la valeur la plus grande.

Continuons notre jeu du pendu en modélisant les trois étapes suivantes par des fonctions :

- Saisie du mot à deviner : une fonction sans paramètre qui renvoie le mot à deviner.
- Saisie de la lettre de l'utilisateur : une fonction sans paramètre qui retourne la lettre saisie par l'utilisateur.
- Tester si le mot a été deviné par l'utilisateur : une fonction qui renvoie un booléen qui indique si le joueur a trouvé le mot ou non.

Remarque

Un sous-programme n'est pas obligé d'avoir des paramètres. Dans ce cas, nous utiliserons une paire de parenthèses vide lors de la déclaration et l'appel pour notifier ce cas.

```
FONCTION saisie_mot() : CHAINE
VAR devinette : CHAINE
DEBUT
   ECRIRE("Entrer le mot à deviner")
   devinette <- LIRE()
   RETOURNER(devinette)
FIN
FONCTION saisie_lettre() : CARACTERE
VAR l : CARACTERE
DEBUT
   ECRIRE("Entrer une lettre")
   l <- LIRE()
   RETOURNER(l)
FIN
FONCTION mot_trouve(masque: CHAINE) : BOOLEEN
VAR i : ENTIER
trouve : BOOLEEN
DEBUT
   trouve <- VRAI
   POUR i ALLANT de 1 à LONGUEUR(mot_clair) AU PAS DE 1
   FAIRE
      SI EXTRAIRE(mot, i, 1) = '_'
      ALORS
         trouve <- FAUX
      FINSI
   FINPOUR
   RETOURNER(trouve)
FIN
```

La fonction déterminant si l'utilisateur a deviné ou non le mot teste l'existence dans le mot masqué du caractère "_". Si ce caractère est trouvé, l'utilisateur n'a pas encore deviné le mot et la fonction retourne faux sinon elle retourne vrai.

Nos fonctions étant déclarées, voyons maintenant comment les appeler.

1.2.2 Appel

On appelle une fonction en précisant son nom et, entre parenthèses, les valeurs que l'on souhaite attribuer aux paramètres dans l'ordre dans lequel ils sont déclarés, comme pour les procédures. Lors de son appel, à la différence d'une procédure, le retour d'une fonction doit être utilisé, en étant stocké dans une variable par exemple :

```
FONCTION maximun(a,b : ENTIER) : ENTIER
DEBUT
   SI (a > b)
   ALORS
      RETOURNER(a)
   SINON
      RETOURNER(b)
   FINSI
FIN
PROGRAMME calcul_maximum
VAR a, b, max : ENTIER
DEBUT
   ECRIRE("Entrez votre premier entier")
   a <- LIRE()
   ECRIRE("Entrez votre second entier")
   b <- LIRE()
   max <- maximun(a, b)
   ECRIRE("Le maximum est : ", max)
FIN
```

Complétons notre programme principal du pendu avec l'appel de nos fonctions :

```
PROGRAMME pendu
VAR mot, masque : CHAINE
lettre : CARACTERE
fin_partie, mot_devine : BOOLEEN
essai : ENTIER
DEBUT
   fin_partie <- FAUX
   essai<- 7
   mot <- saisie_mot()
   TANT QUE NON fin_partie
   FAIRE
      afficher_mot_masque(masque)
      lettre <- saisie_lettre()
```

```
        calcul_masquage(masque, essai, lettre, mot)
        mot_devine <- mot_trouve(masque)
        SI mot_devine = VRAI
        ALORS
            ECRIRE("Gagné")
            fin_partie <- VRAI
        SINON
            SI essai = 0
            ALORS
                ECRIRE("Perdu... Vous deviez deviner : ", mot)
                fin_partie <- VRAI
            FINSI
        FINSI
    FINTANTQUE
FIN
```

Remarque

En pratique, hormis les implémentations du langage SQL, comme par exemple TransactSQL et quelques langages spécifiques, les langages de programmation actuels n'utilisent plus de procédures mais uniquement des fonctions, même si d'un point de vue d'algorithmie ce sont des procédures. Nous verrons plus en détail cette remarque dans la mise en pratique avec Python plus loin dans la section dédiée.

2. L'élégance de la récursivité

Par définition, un sous-programme **récursif** est un sous-programme qui s'appelle lui-même.

L'objectif de la récursivité est de simplifier, en théorie, un traitement complexe contenant des itérations, en écrivant uniquement des traitements simples. Cependant, il existe toujours un fossé entre la théorie et la pratique.

Regardons un peu l'évolution des langages de programmation avec beaucoup de vulgarisation. Les premiers langages de programmation, et encore certains exotiques de nos jours, n'implémentaient pas les structures itératives, seulement les conditionnelles. Comment faire alors pour répéter une instruction ?

Simplement en rappelant le sous-programme contenant cette instruction dans une conditionnelle, jusqu'à ce que la condition d'arrêt de l'itération, mise donc dans la condition du `SI`, soit juste.

Modélisons ce principe avec l'affichage d'un tableau.

Pour afficher un tableau, nous devons le parcourir du premier élément au dernier. Nous devons donc nous arrêter après l'affichage de ce dernier élément. Ainsi notre condition d'arrêt est "nous avons traité le dernier élément".

Un parcours de tableau se fait grâce à un compteur qui s'incrémente au fur et à mesure du parcours. Nous pouvons donc modéliser notre condition d'arrêt ainsi : le compteur doit être supérieur à la taille du tableau. Si j'ai traité le dernier élément du tableau je m'arrête, c'est-à-dire le compteur est arrivé à la taille du tableau.

Il existe plusieurs types de récursivités : la simple, la multiple, la croisée et l'imbriquée. Nous n'aborderons dans ce chapitre que les deux premiers types et nous laissons le lecteur aller plus loin par lui-même pour les deux derniers. Ces différents types de récursivités sont tous basés sur le même principe que la récursivité simple, seule la hiérarchie des appels récursifs change.

2.1 Récursivité simple

La récursivité simple est la plus utilisée. Il s'agit **d'un sous-programme qui s'appelle lui-même pour itérer un traitement**.

Un sous-programme récursif est toujours écrit grâce à une instruction `SI` déterminant la condition d'arrêt avec, dans le `SINON`, l'appel du sous-programme :

```
SOUS-PROGRAMME sous-prog-A
VAR
   ...
DEBUT
   SI condition_arrêt
   ALORS
       ...  // arrêt des appels récursifs
   SINON
       ...
```

```
        sous-prog-A          // appel récursif toujours dans le sinon
    FINSI
FIN
```

Prenons le cas simple d'une fonction qui calcule la somme des n premiers entiers. Pourquoi partir sur une fonction relative aux mathématiques ? Tout simplement car l'explication de l'enchaînement des appels est simplifiée avec les formules commençant notamment par le symbole somme (Σ).

Commençons par écrire la fonction non récursive :

```
FONCTION somme_premiers_entiers(n : ENTIER) : ENTIER
VAR i, somme : ENTIER
DEBUT
   POUR i ALLANT DE 1 à n AU PAS DE 1
   FAIRE
      somme <- somme + i
   FINPOUR
   RETOURNER(somme)
FIN
PROGRAMME calcul_somme
VAR s : ENTIER
DEBUT
   s <- somme_premiers_entiers(4)
   ECRIRE("La somme des 4 premiers entiers est : ", s)
FIN
```

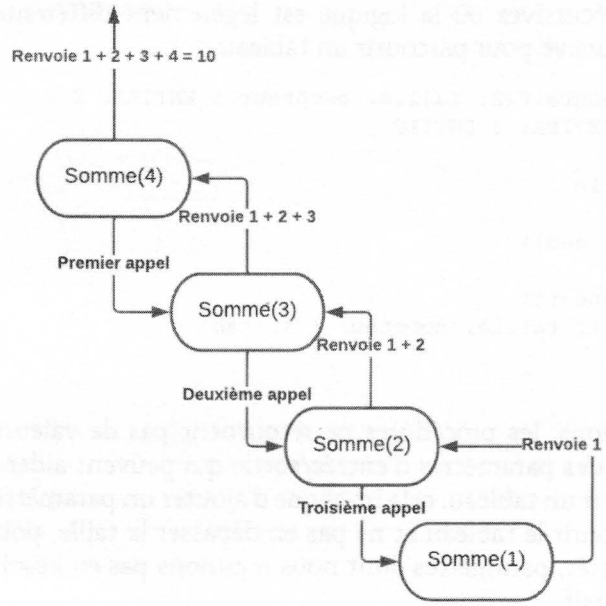
Remarque

Rappel de la règle de la récursivité : toute instruction itérative incluse dans une fonction peut se faire en récursif avec une instruction conditionnelle.

Pour construire cette fonction de manière récursive, nous appliquons la logique suivante :

- La somme des quatre premiers entiers est 1 + 2 + 3 + 4.
- Soit également la somme des trois premiers entiers additionnée de 4.
- Soit encore la somme des deux premiers entiers + (3 + 4).
- Soit la somme du premier entier + (2 + 3 + 4).
- Soit donc finalement 1 + (2 + 3 +4).

L'enchaînement des appels de cette fonction est décrite dans la figure ci-après pour aider le lecteur à en visualiser l'élégance.



Appels récursifs du calcul de la somme des quatre premiers entiers

Cette logique est implémentée par la fonction suivante :

```
FONCTION somme_premiers_entiers_recursive(n : ENTIER) : ENTIER
DEBUT
   SI n = 1
   ALORS
      RETOURNER(1)
   SINON
      RETOURNER(n + somme_premiers_entiers_recursive(n - 1))
   FINSI
FIN
PROGRAMME calcul_somme
VAR s : ENTIER
DEBUT
   s <- somme_premiers_entiers_recursive(4)
   ECRIRE("La somme des 4 premiers entiers est : ", s)
FIN
```

À la lecture, cette fonction récursive est bien plus naturelle pour les humains que la première, la difficulté étant son écriture.

Passons aux procédures récursives où la logique est légèrement différente. Regardons la fonction récursive pour parcourir un tableau :

```
PROCEDURE parcours_recursif(E: taille, compteur : ENTIER, E :
tab : TABLEAU [ ] : ENTIER) : ENTIER
DEBUT
   SI compteur = taille
   ALORS
      ECRIRE(tab[compteur])
   SINON
      ECRIRE(tab[compteur])
      parcours_recursif(taille, compteur + 1, tab)
   FINSI
FIN
```

Contrairement aux fonctions, les procédures ne retournent pas de valeurs. Cependant, elles utilisent des paramètres d'entrée/sortie qui peuvent aider à la récursivité. Pour parcourir un tableau, cela implique d'ajouter un paramètre, ici le compteur pour parcourir le tableau et ne pas en dépasser la taille, pour modéliser la condition d'arrêt, paramètres dont nous n'aurions pas eu besoin avec un parcours non récursif.

Nous pouvons remarquer que la lecture de cette procédure n'est pas forcément plus naturelle, contrairement à la fonction itérative. Cela vient du fait du retour qu'impose la récursivité.

Augmentons maintenant légèrement la difficulté avec la récursivité multiple.

2.2 Récursivité multiple

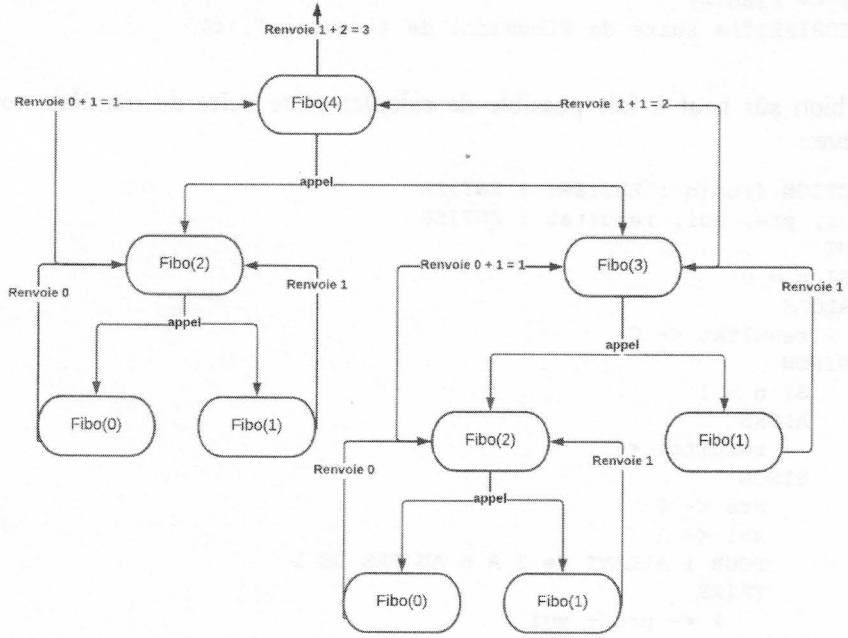
La récursivité multiple représente un sous-programme qui s'appelle plusieurs fois dans la même instruction. Elle n'a rien de plus compliqué que la récursivité simple.

Le plus simple exemple de récursivité multiple, et qui est aussi un grand élément de culture à connaître en informatique, est la suite de Fibonacci. Chaque élément n de la suite est notée Fn et est défini par les calculs suivants :

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ pour tout n supérieur à 1.

En français, cette suite retourne la somme des derniers termes de la suite avec les cas particuliers du 0 et du 1 qui retournent leur propre valeur. Ces deux termes sont donc nos conditions d'arrêts.

L'appel récursif de la suite est littéralement donné dans la définition de la suite : pour calculer un élément, nous devons calculer les deux précédents, donc les deux précédents de ses derniers, etc. comme représenté dans la figure ci-après :



Appels récursifs pour la suite de Fibonacci

Cette logique peut se traduire par la fonction suivante :

```
FONCTION fibo(n : ENTIER) : ENTIER
DEBUT
   SI n = 0
   ALORS
      RETOURNER(0)
   SINON
      SI n = 1
      ALORS
         RETOURNER(1)
      SINON
         RETOURNER(fibo(n - 1) + fibo(n - 2))
      FINSI
   FINSI
FIN
PROGRAMME calcul_fibo
VAR f : ENTIER
DEBUT
   f <- fibo(4)
   ECRIRE("La suite de Fibonacci de 4 vaut : ", f)
FIN
```

Il est bien sûr tout à fait possible de calculer cette suite de manière non récursive :

```
FONCTION fibo(n : ENTIER) : ENTIER
VAR i, pre, sui, resultat : ENTIER
DEBUT
   SI n = 0
   ALORS
      resultat <- 0
   SINON
      SI n = 1
      ALORS
         resultat <- 1
      SINON
         pre <- 0
         sui <- 1
         POUR i ALLANT de 2 A n AU PAS DE 1
         FAIRE
            k <- pre + sui
            pre <- sui
            sui <- k
```

```
        FINPOUR
    FINSI
  FINSI
  RETOURNER(resultat)
FIN
PROGRAMME calcul_fibo
VAR f : ENTIER
DEBUT
  f <- fibo(4)
  ECRIRE("La suite de Fibonacci de 4 vaut : ", f)
FIN
```

Nous remarquons facilement que la fonction récursive est plus simple à lire que la fonction non récursive.

Elle est également plus simple, ou logique, à écrire car elle demande moins de réflexion : nous appliquons la définition telle quelle.

Nous pouvons alors nous poser la question : quand devons-nous utiliser des itérations et quand devons-nous utiliser la récursivité ?

2.3 Itération ou récursivité ?

Il va de soi que chaque structure itérative ne doit pas être remplacée par un sous-programme récursif, sinon pourquoi avoir ajouté les itérations aux langages de programmation ?

Malgré son élégance, la récursivité peut être dangereuse : si la condition d'arrêt est mal déterminée, alors le programme bouclera à l'infini et ne se terminera jamais. De plus, les appels récursifs peuvent également saturer la mémoire de l'ordinateur dans certains cas, donc planter l'exécution du programme par débordement de la mémoire.

Nous pouvons toujours transformer un sous-programme récursif en un sous-programme non récursif mais certains peuvent s'avérer extrêmement compliqués.

L'exemple classique pour illustrer ce propos est le jeu du démineur. Lorsque l'utilisateur clique sur une case non minée et dont les voisines ne sont pas minées, la découverte des cases voisines se propagent jusqu'à atteindre une case voisine d'une mine ou le bord du plateau.

En non récursif, il nous faudrait écrire des structures conditionnelles qui permettent de naviguer dans les huit sens sur la matrice représentant le plateau de jeu et s'arrêter uniquement lorsque nous atteignons une case voisine d'une mine sans dépasser de la matrice et ce, sans oublier toutes les autres cases voisines à dévoiler. Une imbrication extrêmement complexe de structures itératives et conditionnelles...

En récursif, la logique est bien plus naturelle : il nous suffit de définir un sous-programme "dévoiler" qui s'appellera sur les huit cases voisines lorsqu'il est appelé sur une case sans mine et non voisine d'une mine. Ainsi la propagation du dévoilement des cases à dévoiler se fait naturellement sans réfléchir aux cases que nous traitons sur la matrice. La gestion des bords du plateau se fait dans ses sous-programmes en faisant partie de la condition d'arrêt.

Nous étudierons ce jeu plus en détail et de manière pratique dans les exercices de ce chapitre. Nous verrons également au chapitre Les fichiers que la manipulation des structures complexes de données est plus facile en récursif.

En attendant revenons aux tableaux.

3. Algorithmes avancés sur les tableaux

L'objectif principal de l'informatique est de simplifier les traitements complexes pour l'utilisateur. Il va donc de soi que lorsqu'un programme travaille sur des tableaux, saisis par l'utilisateur ou non, il doive trier ces tableaux pour en faciliter les manipulations et donc les recherches.

Maintenant que nous avons étudié les sous-programmes et la récursivité, nous pouvons voir comment effectuer ces tris en partant des algorithmes les plus simples pour l'être humain, donc les moins performants pour la machine, aux plus optimisés pour l'ordinateur, donc les moins simples pour le cerveau humain (sous-entendu utilisant des sous-programmes récursifs).

3.1 Procédure échanger

Lorsque nous trions un tableau, nous sommes sûrs de devoir échanger des valeurs assez fréquemment. Nous allons définir une procédure pour effectuer cette opération afin de la réutiliser dans les algorithmes de tri qui suivent.

```
PROCEDURE echanger(E/S : x, y : ENTIER)
VAR
   tmp : ENTIER
DEBUT
   tmp <- x
   x <- y
   y <- tmp
FIN
```

3.2 Tri par sélection

Le tri par sélection est le tri ayant la logique la plus simple. Nous commençons par chercher la plus petite valeur du tableau et nous la plaçons à la première case en décalant les autres cases, comme le montre la figure ci-après. Il ne nous reste plus qu'à appliquer les mêmes opérations sur le sous-tableau commençant à la case 2, puis celui commençant à la case 3, etc. jusqu'à arriver à un dernier sous-tableau de taille 1, ce qui indique que le tableau est bien trié.

4	9	3	10	4	On cherche la plus petite valeur : 3
3	9	4	10	4	On la permute avec la première valeur
3	9	4	10	4	On cherche la plus petite valeur : 4
3	4	9	10	4	On la permute avec la deuxième valeur
3	4	9	10	4	On cherche la plus petite valeur : 4
3	4	4	10	9	On la permute avec la troisième valeur
3	4	4	10	9	On cherche la plus petite valeur : 9
3	4	4	9	10	On la permute avec la quatrième valeur

Notre tableau est trié !

Tri par sélection

Le tri par sélection demande donc deux manipulations :

- **Trouver la plus petite valeur**.
- **Échanger** la case courante avec la première case du sous-tableau correspondant.

Nous avons donc besoin d'un sous-programme pour rechercher la plus petite valeur, ou plutôt l'indice de la plus petite valeur d'un tableau.

```
FONCTION indice-minimum (tab : TABLEAU[] : ENTIER, début,
taille : ENTIER) : ENTIER
VAR
   i, indice : ENTIER
DEBUT
   indice <- début
   POUR i ALLANT DE (début + 1) à taille AU PAS DE 1
   FAIRE
      SI (tab[i] < tab[indice ])
      ALORS
```

```
            Indice <- i
        FINSI
    FINPOUR
    RETOURNER(indice)
FIN
```

Nous n'avons plus qu'à définir l'échange entre la première case et la plus petite valeur pour chaque sous-tableau :

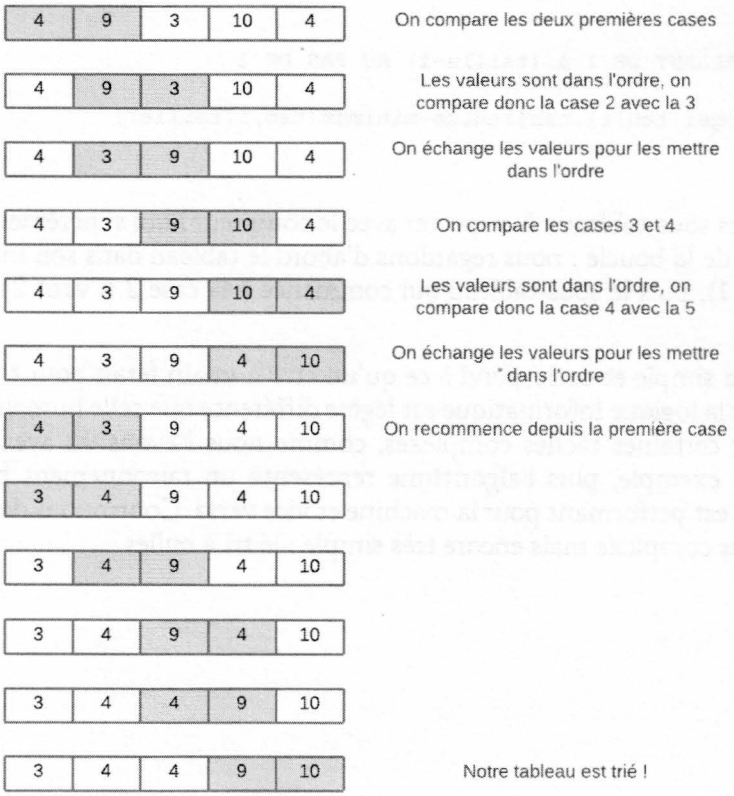
```
PROCEDURE tri-selection (E/S : tab : TABLEAU[] : ENTIER, E :
taille : ENTIER)
VAR
i : ENTIER
DEBUT
    POUR i ALLANT DE 1 à (taille-1) AU PAS DE 1
    FAIRE
        echanger(tab[i],tab[indice-minimum(tab,i,taille))
    FINPOUR
FIN
```

Nous gérons les sous-tableaux à inspecter avec le compteur i qui s'incrémente à chaque tour de la boucle : nous regardons d'abord le tableau dans son intégralité (i vaut 1), puis le sous-tableau qui commence à la case 2 (i vaut 2) et ainsi de suite.

Ce tri est assez simple et correspond à ce qu'un être humain ferait pour trier un tableau. Or la logique informatique est légère différente que celle humaine. En effet, pour certaines tâches complexes, comme nous l'avons vu avec la récursivité par exemple, plus l'algorithme représente un raisonnement humain, moins il est performant pour la machine et vice versa. Continuons donc avec un tri plus complexe mais encore très simple : le tri à bulles.

3.3 Tri à bulles

Le tri à bulles parcourt le tableau à trier en comparant chaque élément avec son successeur, comme le montre la figure ci-dessous. Cette paire d'éléments du tableau est la **bulle**, qui va se déplacer dans le tableau au fur et à mesure du tri. Dans la bulle, le tri place la plus petite valeur en premier, la plus grande en deuxième. Le tri par bulles répète le parcours entier du tableau jusqu'à qu'il n'y ait aucune permutation au sein d'une bulle, ce qui indique que le tableau est trié.



Tri à bulles

Ayant déjà la procédure pour échanger deux valeurs, donc pour permuter les deux éléments de la bulle, il ne nous reste qu'à écrire la procédure correspondant au tri à bulles.

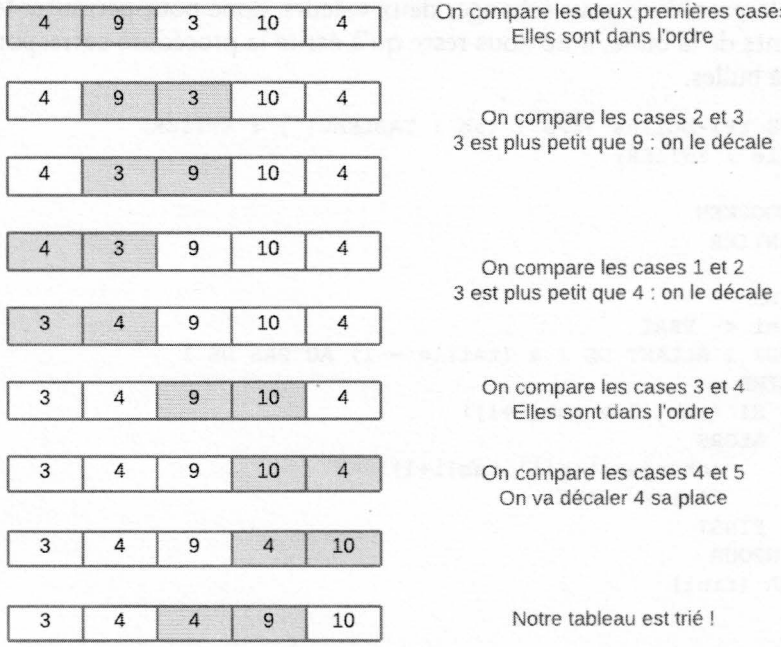
```
PROCEDURE tri-bulles (E/S : tab : TABLEAU[ ] : ENTIER,
E : taille : ENTIER)
VAR
fini : BOOLEEN
   i : ENTIER
DEBUT
   REPETER
      fini <- VRAI
      POUR i ALLANT DE 1 à (taille - 1) AU PAS DE 1
      FAIRE
         SI (tab[i] > tab[i+1])
         ALORS
            echanger(tab[i],tab[i+1])
            fini <- FAUX
         FINSI
      FINPOUR
   JUSQUA (fini)
FIN
```

La logique de cet algorithme est relativement simple : nous permutons deux éléments s'ils ne sont pas dans l'ordre et nous arrêtons suite à un parcours sans permutation.

Le défaut majeur de ce tri est le nombre de parcours effectués sur le tableau pour le trier. Pour pallier ce défaut, il existe le tri par insertion.

3.4 Tri par insertion

Dans le tri par insertion, nous commençons par le découper en sous-tableaux commençant toujours à la première case : le premier de taille 1, le deuxième de taille 2, le troisième de taille 3..., le dernier de la taille du tableau.



Tri par insertion

Dans chacun des sous-tableaux, nous ordonnons correctement leurs éléments dans l'ordre, comme le montre la figure ci-dessus.

Avec ce tri, nous nous basons toujours sur un tableau trié auquel nous ajoutons une seule valeur, ce qui simplifie énormément le placement de cette nouvelle valeur au sein du tableau.

```
PROCEDURE tri-insertion (E/S : tab : TABLEAU[] : ENTIER,
E : taille : ENTIER)
VAR
   i, tmp, j : ENTIER
DEBUT
   POUR i ALLANT DE 2 à taille AU PAS DE 1
   FAIRE
      tmp <- tab[i]
      j <- i - 1
      TANT QUE ((j > 0) ET (tab [j] > tmp))
      FAIRE
```

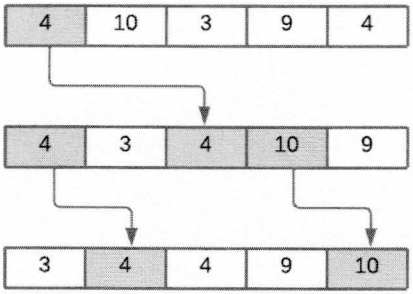
```
            tab [j+1] <- tab [j]
            j <- j - 1
        FINTANTQUE
        tab [j+1] <- tmp
    FINPOUR
FIN
```

Le premier sous-tableau étant composé d'une seule case, il est déjà trié. Nous commençons notre structure itérative avec un compteur commençant à 2.

Pour chaque sous-tableau à traiter, nous le parcourons du dernier élément au premier en décalant les plus petites valeurs à la case précédente au besoin.

3.5 Tri rapide

Le tri rapide est le tri généralement implémenté dans les langages de programmation de haut niveau. Il est illustré sur la figure ci-dessous. Il commence par prendre la première valeur du tableau qui sera le **pivot**. Il place les valeurs plus petites que le pivot à sa gauche dans le tableau et les valeurs plus grandes à sa droite.



Tri rapide

Il va par la suite répéter le choix du pivot et le placement des éléments pour la partie avant le premier pivot et la partie après le premier pivot. Il s'arrête lorsqu'il n'y a plus d'éléments à gauche et à droite du pivot courant.

Avec cette description, nous nous rendons rapidement compte que ce tri utilise la récursivité en se rappelant sur les deux parties du tableau à traiter séparées par la case pivot.

Ce tri demande plusieurs traitements donc plusieurs sous-programmes :

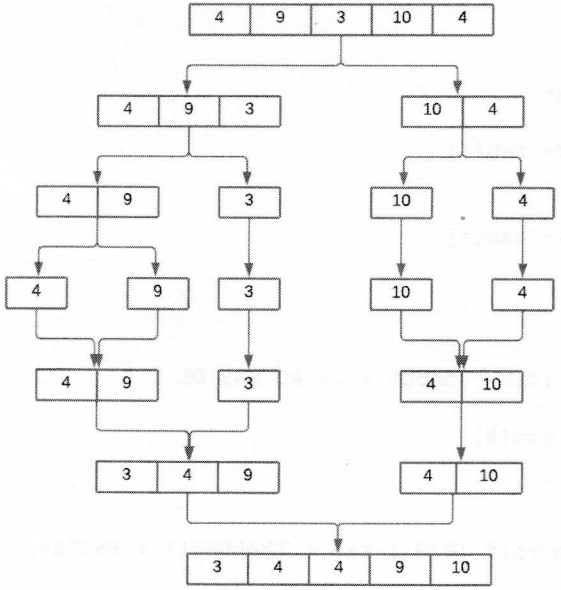
- Déterminer le pivot.
- Appliquer le tri récursif.
- Lancer le tri.

```
PROCEDURE placer-pivot (E/S : tab : TABLEAU[] : ENTIER,
E : taille, début, fin : ENTIER, S : indice_pivot : ENTIER)
VAR
   i, j, pivot : ENTIER
DEBUT
   pivot <- tab[début]
   i <-début
   j <- fin
   TANTQUE (i <= j)
   FAIRE
      TANTQUE ( (tab[i] <= pivot) ET (i <= j) )
      FAIRE
         i <- i + 1
      FINTANTQUE
      TANTQUE ( (tab[j] > pivot) ET (i <= j) )
      FAIRE
         j <- j - 1
      FINTANTQUE
      SI (i <= j)
      ALORS
         echanger(tab[i],tab[j])
      FINSI
   FINTANTQUE
   indice_pivot <- j
   echanger(tab[début],tab[j])
FIN

PROCEDURE tri-rapide-recursif (E/S : tab : TABLEAU[] : ENTIER,
E : taille, début, fin : ENTIER)
VAR indice_pivot : ENTIER
DEBUT
   Si (début < fin)
   ALORS
      placer-pivot(tab,taille,début,fin,indice_pivot)
      tri-rapide-recursif(tab,taille,début,indice_pivot-1)
      tri-rapide-recursif(tab,taille,indice_pivot+1,fin)
   FINSI
```

```
FIN

PROCEDURE TriRapide (E/S : tab : TABLEAU[] : ENTIER, E : taille)
DEBUT
   tri-rapide-recursif(tab,taille,1,taille)
FIN
```

3.6 Tri fusion

Le tri par fusion est le tri le plus rapide en informatique. Il **scinde** le tableau à trier en deux sous tableaux de **même taille**, à un élément près si sa taille est impaire. Il effectue cette séparation jusqu'à n'avoir que des **tableaux d'une case**. Ensuite, il va **regrouper les cases** des tableaux intermédiaires en rangeant les valeurs dans l'ordre comme le montre la figure ci-dessous :



Tri fusion

Nous voyons également avec ce tri qu'il possède un raisonnement récursif semblable à celui du tri rapide.

```
PROCEDURE fusionner(E/S : tab : TABLEAU[] : ENTIER, E :
taille, début, milieu, fin : ENTIER)
VAR i, j ,k : ENTIER
   tab_tmp : TABLEAU[1…taille] : ENTIER
DEBUT
   i <- début
   j <- fin
   POUR k ALLANT DE 1 à (fin - début + 1) AU PAS DE 1
   FAIRE
      SI ( (i <= milieu) ET (j <= fin) )
      ALORS
         SI (tab[i] <= tab[j])
         ALORS
            tab_tmp[k] <- tab[i]
            i <- i + 1
         SINON
            tab_tmp[k] <- tab[j]
            j <- j + 1
         FINSI
      SINON
         SI (i <= milieu)
         ALORS
            tab_tmp[k] <- tab[i]
            i <- i + 1
         SINON
            tab_tmp[k] <- tab[j]
            j <- j + 1
         FINSI
      FINSI
   FINPOUR
   POUR k ALLANT DE 1 à (fin - début + 1) AU PAS DE 1
FAIRE
      tab[début+k-1] <- pro[k]
   FINPOUR
FIN

PROCEDURE tri-fusion-recursif (E/S : tab : TABLEAU[] : ENTIER,
E : taille, début, fin)
DEBUT
   SI (début < fin)
   ALORS
      tri-fusion-recursif(tab,taille,début,(début+fin) DIV 2)
      tri-fusion-recursif(tab,taille,(début+fin) DIV 2,fin)
      fusionner(tab,taille,début,(début+fin) DIV 2,fin)
```

```
    FINSI
FIN
PROCEDURE triFusion (E/S : tab : TABLEAU[] : ENTIER, E : taille)
DEBUT
    triFusion(tab,1,taille)
FIN
```

Remarque

De nos jours, le lecteur n'a plus à apprendre par cœur comment implémenter les algorithmes de tri car ils sont implémentés au sein des langages de programmation. Cependant, comprendre et être capable de reproduire permettra au lecteur d'avoir un meilleur niveau en programmation car vous aurez acquis un niveau d'abstraction vous permettant de comprendre le fonctionnement du langage avec lequel vous codez.

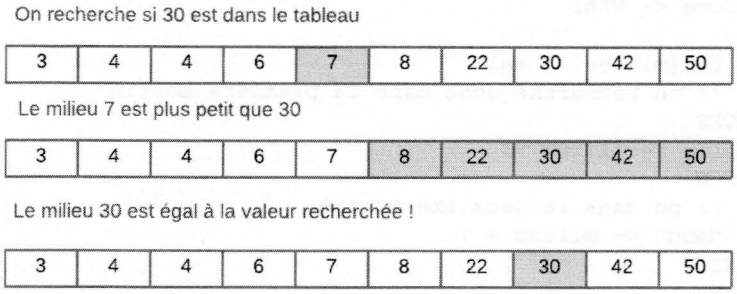
Remarque

La fonction `sort` de Python implémente un tri hybride mélangeant le tri par fusion et le tri par insertion, appelé le ***Timsort****. Nous vous laissons étudier ce tri par vous-même maintenant que vous avez les bases sur les deux tris utilisés.*

3.7 Recherche dichotomique

Maintenant que nous pouvons trier nos tableaux, nous pouvons optimiser la recherche d'une valeur dans ces derniers.

Pour ce faire, nous pouvons utiliser une recherche dichotomique, comme illustrée dans la figure suivante.



Recherche dichotomique

Le principe d'une telle recherche est assez simple. Nous commençons par regarder la valeur de la case se trouvant au milieu du tableau. Dans le meilleur des cas, il s'agit de la valeur recherchée. Si ce n'est pas le cas, si elle est plus grande que la valeur recherchée, nous regardons alors dans la première moitié du tableau. Si elle est plus petite, nous regardons dans la seconde moitié du tableau Nous répétons ces opérations jusqu'à trouver la valeur. Dans le cas où la valeur recherchée n'est pas dans le tableau, notre dernière moitié de tableau sera composée d'une seule case qui ne sera pas la valeur recherchée.

En algorithme, la séparation du tableau en deux parties se base sur la valeur de plusieurs compteurs : un pour l'indice de départ, un pour l'indice de fin et un pour l'indice du milieu. Nous allons changer la valeur de ces compteurs pour indiquer dans quelle moitié rechercher la valeur demandée.

```
PROGRAMME Recherche_dichotomique
VAR
   tab : TABLEAU[1…10] : ENTIER
   debut, milieu fin, valeur : ENTIER
   trouve <- FAUX : BOOLEEN
DEBUT
   // initialisation et tri du tableau déjà effectués
   ECRIRE("Entrer la valeur à rechercher")
   val <- LIRE()
   debut <- 1
   fin <- 10    // la taille du tableau
   TANTQUE debut <= fin ET NON trouve
   FAIRE
      milieu <- (debut + fin ) DIV 2
      SI tab[milieu] = val
      ALORS
         trouve <- VRAI
      SINON
         SI tab[milieu] < val
            // on recherche donc dans la première moitié
         ALORS
            fin <- milieu -1
         SINON
            // ou dans la deuxième moitié
            debut <- milieu + 1
         FINSI
         FINSI
   FINTANTQUE
```

```
    SI trouve
    ALORS
       ECRIRE("la valeur est présente dans le tableau)
    SINON
       ECRIRE("le tableau ne contient pas la valeur demandée")
    FINSI
FIN
```

Remarque

Vous pouvez remarquer que nous avons juste testé la valeur des variables de type booléen dans nos conditions. Cela revient au même que de faire une comparaison booléenne.

4. Fonctions et procédures avec Python

Comme nous l'avons remarqué dans la section précédente, le Python n'implémente que les fonctions.

4.1 Les fonctions en Python

Les fonctions en Python sont obligatoirement **déclarées en début de script**. Le bloc d'instructions est obligatoire pour les fonctions. Si nous devons, pour une raison de praticité, laisser le corps d'une fonction vide, nous utilisons le mot-clé `pass`. Pour définir une fonction, nous utilisons le mot-clé `def`.

Une bonne pratique pour séparer les fonctions du code principal est d'utiliser une instruction `if` particulière à Python :

```
if __name__ == '__main__':
```

```
# une fonction sans corps
def maFontion():
   pass

# code pour la procédure affiche_tableau
def afficheTableau(tab):
   for e in tab:
      print(e)
```

```
# code de la procédure double
def double(a):
   return a * a

# code de la fonction maximun
def maximun(a,b):
   if a > b :
      return a
   else:
      return b

# programme principal
if __name__ == '__main__':
   afficheTableau([1,2,3])
   a = 2
   aa = double(2)
   print("le double de", a, "est", aa)
   print("le maximun est", maximum(2,3))
```

Ce code reprend les procédures et fonctions définies dans les sections précédentes. Nous pouvons remarquer plusieurs différences entre l'algorithmie et le Python :

- Le type de retour n'est pas déclaré car Python est un langage typé dynamiquement.
- L'instruction `RETOURNER` est remplacée par l'instruction `return`.
- Une procédure sans sortie est une fonction sans return.
- Une fonction retournant une valeur peut être directement appelée dans une instruction `print`.
- Nul besoin de passer la taille d'un tableau lorsque ce dernier est passé en paramètre.

4.2 Particularités de Python

Comme tout langage de programmation, Python a ses spécificités propres, notamment au niveau des fonctions, qui ne sont pas forcément implémentées dans les autres langages.

Retour multiple

En Python, une fonction peut **retourner plusieurs valeurs**, ce que nous appelons un retour multiple. Pour ce faire, il suffit de séparer les différentes valeurs par des virgules dans l'instruction `return`.

Remarque

Attention à ne mettre pas les retours multiples entre parenthèses pour une question d'esthétisme, cela reviendrait à faire un retour simple d'un tuple et non de plusieurs valeurs.

Lors de l'appel d'une fonction avec retour multiple, nous récupérons plusieurs valeurs. Il nous faudra donc énumérer ses valeurs avant l'affectation en les séparant par des virgules.

```
# la procédure de calculs sur un rectangle
def calculRectangle(la, lo):
    aire = la * lo
    perimetre = la * 2 + lo * 2
    return aire, perimetre

if __name__ == '__main__':
    la = 2
    lo = 4
    aire, perimetre = calculRectangle(la,lo)
    print("l'aire du rectangle est", aire)
    print("le périmtère du rectangle est", perimetre)
```

Remarque

Attention à l'ordre des valeurs du retour qui doit être respecté lors de l'affectation avec l'appel de la fonction.

Valeurs par défaut

Python permet de donner une **valeur par défaut** aux paramètres en les affectant lors de la déclaration de la fonction.

```
# maximun entre trois valeurs
def maximum3(a=0, b=1, c=3):
    if a > b  and a > c:
        return a
    elif b > c:
        return b
    else:
        return c

if __name__ == '__main__':
    print("appel avec les valeurs par défaut", maximum3())
    print("appel sans les valeurs par défaut", maximum3(1, 2, 3))
    print("appel avec quelques valeurs par défaut", maximum3(3, 4))
# a vaut 3, b 4 et c 3
    print("appel avec quelques valeurs par défaut", maximum3(4))
# a vaut 0, b 4 et c 3
    print("appel avec quelques valeurs par défaut",
maximum3(b=3, c=4)) # a vaut 0, b 3 et c 4
```

Nous pouvons tout à fait appeler la fonction `maximum3` définie précédemment sans paramètre, avec certains paramètres ou avec tous les paramètres.

Lors d'un appel sans paramètre, ce sont uniquement les valeurs par défaut qui seront affectées à tous les paramètres.

Lorsque nous appelons la fonction avec autant de valeurs que de paramètres, aucune valeur par défaut n'est utilisée.

Si nous sommes dans l'obligation d'utiliser la valeur par défaut d'un paramètre autre que le dernier (ou les derniers), nous devons impérativement nommer le ou les paramètres qui n'utilisent pas les valeurs par défaut comme le montre l'extrait de code précédent. Sans cela, Python ne saura pas à quel paramètre affecter les valeurs.

Lambda expression

Pour des fonctions simples, ne nécessitant qu'une instruction `return` avec un calcul, Python nous propose de les implémenter dans des lambda expressions. Ces expressions peuvent être déclarées n'importe où dans le code, comme des variables. Leur appel ne change pas de celui de fonctions classiques.

Une expression lambda est **déclarée par un identifiant auquel nous affectons la fonction grâce au mot-clé** `lambda`. Les paramètres suivent le mot-clé `lambda` sans parenthèses et le retour est indiqué par les deux points.

```
if __name__ == '__main__':
    double = lambda a : a * 2
    print("le double de 2 est", double(2))
```

Les fonctions lambda doivent être uniquement utilisées pour des calculs très simples d'une seule instruction afin de ne pas en surcharger la lecture et donc la compréhension.

5. Exercices

5.1 Exercice 1

Écrivez un algorithme qui calcule dans un sous-programme la surface d'un cercle. Codez le script Python correspondant.

5.2 Exercice 2

Écrivez un algorithme qui calcule dans un sous-programme le volume d'une boîte. Codez le script Python correspondant en utilisant des valeurs par défaut pour les paramètres de la fonction.

5.3 Exercice 3

Écrivez un algorithme qui calcule et retourne dans un sous-programme la valeur la plus grande d'un tableau entre celles des deux indices passés en paramètres. Codez le script Python correspondant.

5.4 Exercice 4

Écrivez un algorithme avec un sous-programme qui retourne le nom du mois en fonction de son numéro. Par exemple, si l'utilisateur entre 1, il sera affiché janvier. Codez le script Python correspondant.

5.5 Exercice 5

Écrivez un algorithme qui calcule le nombre de mots d'une chaîne de caractères avec un sous-programme. Nous considérons que deux mots sont uniquement séparés par un espace pour des raisons de simplicités. Codez le script Python correspondant.

5.6 Exercice 6

Écrivez un algorithme qui donne le nombre de chiffres d'un entier entré par l'utilisateur. Codez le script Python correspondant.

5.7 Exercice 7

Écrivez un algorithme qui détermine si deux mots sont des anagrammes ou non. Codez le script Python correspondant.

5.8 Exercice 8

Écrivez un algorithme qui calcule dans un sous-programme la distance de Hamming entre deux paramètres de type chaînes de caractères. Cette distance représente le nombre de caractères différents entre les deux chaînes de même taille. Par exemple, la distance entre chien et niche est 5 et celle entre bob et sos de 2. Codez le script Python correspondant.

5.9 Exercice 9

Écrivez l'algorithme entier puis le code Python pour le jeu du morpion en le découpant en sous-programme :

- `initGrille(grille)` : grille.
- `grilleComplete(grille)` : boolean.
- `winner(grille)` : boolean.
- `afficheGrille(grille)`.
- `saisirJoueur(grille)` : int,int (un joueur ne doit pas pouvoir tricher et il doit rentrer un int entre 0 et 2 compris).

5.10 Exercice 10

Codez en Python le tri à bulles.

5.11 Exercice 11

Implémentez dans un script Python la recherche dichotomique.

5.12 Exercice 12 - Récursivité

Implémentez dans un script Python le démineur en le décomposant en sous-programmes. Pensez à écrire les algorithmes des sous-programmes avant.

Chapitre 7
Passons en mode confirmé

1. Les pointeurs et références

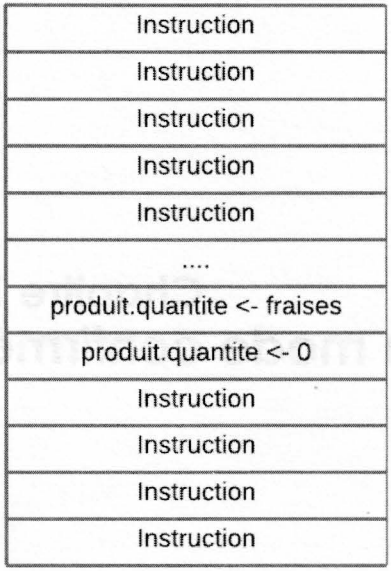
1.1 Implémentation de la mémoire

Lorsque nous exécutons un programme sur un ordinateur, ce dernier doit stocker toutes les informations de ce programme pour le faire tourner correctement. Pour stocker les données en cours d'exécution, il utilise une zone mémoire qui lui est dédiée, sa **pile** (ou *stack* en anglais).

La pile fonctionne comme une pile d'objets dans la vraie vie : nous pouvons récupérer un objet de la pile en commençant par l'objet du dessus, donc le dernier à être mis dans la pile. Le premier objet de la pile est donc le dernier que nous pouvons récupérer.

La pile ne peut pas se souvenir d'un élément précis vu qu'ils sont dépilés, donc sortis de la pile, au fur et à mesure de l'exécution du programme. Ce système permet à la pile d'être extrêmement rapide pour exécuter les instructions car justement elle ne retient rien.

Nous pouvons voir cette pile comme un espace compact où chaque case possède une taille fixe en octets et surtout avec un nombre de cases fixe pour simplifier son implémentation.



Pile utilisée lors de l'exécution d'un programme

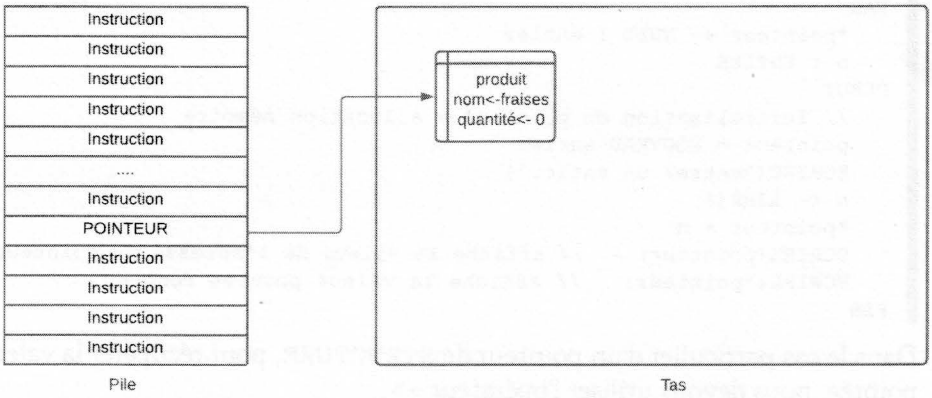
Prenons un exemple vulgarisé avec notre structure de données représentant les produits d'une liste de course. Nous voyons que sur la figure ci-dessus, cette structure va nous prendre deux cases. Or, ces cases ne représentent-elles pas une seule variable ?

1.2 Gestion des pointeurs

Pour économiser les cases de la pile et pouvoir retrouver des éléments précis, nous avons la possibilité de stocker des instructions dans le **tas** (ou *heap* en anglais), une autre zone mémoire dédiée pour l'exécution d'un programme.

Le tas est utilisé pour les allocations de mémoire dynamique : n'importe quel bloc de données peut être alloué ou désalloué n'importe quand. Sa gestion est donc plus complexe mais elle permet de stocker nos structures complexes de données proprement.

Pour stocker des valeurs dans le tas, il nous faut utiliser un **pointeur,** comme le montre la figure ci-dessous et ainsi n'occuper qu'une case de la pile.



Pile, tas et pointeur

Un pointeur se déclare avec une **étoile** devant le nom de la variable. Un pointeur possède deux propriétés : **l'adresse mémoire et la valeur pointée**.

```
VAR
   *mon_pointeur_entier : ENTIER
```

Pour utiliser un pointeur, il faut lui allouer de la mémoire grâce à l'opérateur `NOUVEAU` suivi du type de la valeur pointée. Si nous voulons désallouer l'espace mémoire d'un pointeur, il suffit de l'affecter à la valeur particulière `NULL`, qui indique que les blocs du tas ne sont plus utilisés et peuvent donc être détruits.

Tout pointeur qui a été alloué dans un programme doit être désalloué avant la fin de ce programme pour éviter les fuites mémoire. Une fuite mémoire peut provoquer de graves problèmes comme une extrême lenteur du programme ou même du système d'exploitation entier.

Pour récupérer une valeur simple pointée, nous devons utiliser l'opérateur *.

```
PROGRAMME Exemple_pointeur
VAR
   *pointeur <- NULL : entier
   n : ENTIER
DEBUT
   // Initialisation du pointeur = allocation mémoire
   pointeur = NOUVEAU entier
   ECRIRE("Entrer un entier")
   n <- LIRE()
   *pointeur = n
   ECRIRE(pointeur)     // affiche la valeur de l'adresse du pointeur
   ECRIRE(*pointeur)    // affiche la valeur pointée donc n
FIN
```

Dans le cas particulier d'un pointeur de STRUCTURE, pour récupérer la valeur pointée, nous devons utiliser l'opérateur ->.

```
PROGRAMME Exemple_pointeur_de_structure
STRUCTURE produit
DEBUT
   nom : CHAINE
   quantité : ENTIER
FINSTRUCTURE
VAR
   *pointeur <- NULL : produit
DEBUT
   // Initialisation du pointeur = allocation mémoire
   pointeur = NOUVEAU produit
   ECRIRE("Entrer un nom de produit")
   pointeur->nom <- LIRE()
   ECRIRE("Entrer la quantité de ce produit")
   pointeur->quantite <- LIRE()
   ECRIRE(pointeur->nom, " : ", pointeur->quantite)
FIN
```

Les pointeurs sont essentiels aux structures de données complexes couramment utilisées en informatique car ils permettent une bonne gestion de la mémoire de l'ordinateur lors de l'exécution d'un programme.

2. Les listes chaînées

Les tableaux ont comme limite leur **taille fixe** : soit nous connaissons en avance les dimensions exactes du tableau et nous sommes certains qu'elles ne changeront jamais, soit, dans le doute, nous lui attribuons des dimensions plus grandes que nécessaire pour anticiper les modifications futures, quitte à perdre de l'espace mémoire (les cases non utilisées du tableau).

Pour pallier le problème de la taille fixe des tableaux, il existe des structures complexes de données, comme les **listes**. Les listes ont une **allocation mémoire dynamique** : leur taille augmente à la demande du développeur.

Une liste est un ensemble de "**maillons**", liés entre eux par un **pointeur** et contenant une valeur, équivalente à la case d'un tableau à une dimension.

Remarque

La plupart des langages de programmation implémentant ces structures complexes, le lecteur a tout avantage à en connaître le fonctionnement afin de mieux comprendre ce qui est codé au sein du langage, donc de mieux programmer et de programmer de manière plus efficiente.

2.1 Listes simplement chaînées

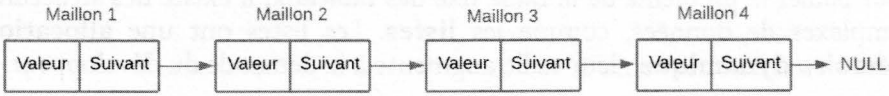
Les premières listes sont les listes **simplement chaînées**. Ce type de liste ne peut être lu **qu'à partir du premier maillon jusqu'au dernier**.

Contrairement aux tableaux, quel que soit le type de la liste, l'opérateur d'indexation `[]` n'existe pas. L'accès à une liste est uniquement **séquentiel** : chaque maillon permet d'aller au suivant, toujours en commençant par le premier.

2.1.1 Création

Pour créer une liste simplement chaînée, nous devons d'abord représenter ce qu'est un maillon. Pour ce faire, nous allons utiliser une `STRUCTURE` contenant deux champs, représentée dans la figure ci-dessous :

- La **valeur** de la donnée.
- Le **pointeur** vers le maillon suivant.



Représentation d'une liste simplement chaînée

Pour indiquer que le maillon est le dernier de la liste, le champ suivant pointera sur la valeur `NULL`.

```
STRUCTURE maillon
DEBUT
   valeur : type
   *suivant <- NULL : maillon
FINSTRUCTURE
```

Tout comme les tableaux, les listes ne peuvent contenir que des valeurs de même type comme le montre la `STRUCTURE maillon`. Nous partons du principe que chaque maillon créé sera le dernier, d'où l'initialisation du pointeur suivant à `NULL`, c'est-à-dire le dernier élément de la liste.

Grâce à la `STRUCTURE maillon`, nous pouvons maintenant définir la `STRUCTURE liste`, représentée sur la figure vue précédemment :

```
STRUCTURE liste
DEBUT
    taille : ENTIER
    *premier <- NULL : maillon
  FINSTRUCTURE
```

```
PROGRAMME Declaration_liste
VAR
   Ma_liste : liste
DEBUT
FIN
```

En plus de contenir son premier maillon, une liste possède également une taille, c'est-à-dire le nombre de maillons. Cela nous permettra de la manipuler plus facilement par la suite.

Toute la difficulté des listes réside dans la bonne gestion des pointeurs représentant le lien entre leurs maillons.

2.1.2 Parcours

Pour manipuler une liste, il nous faut définir le type manipulé par les maillons de cette liste dans la `STRUCTURE maillon`. Nous choisissons de travailler sur des entiers pour illustrer toutes les manipulations des listes simplement chaînées.

Dans cette section, nous demandons au lecteur de considérer une liste d'entiers déjà initialisée, car nous ne nous préoccupons que de son parcours.

Le principe du parcours d'une liste est le suivant : tant que le maillon courant n'est pas à `NULL`, nous affichons sa valeur et nous allons au maillon suivant. Nous commençons bien sûr par le premier maillon, le seul accès aux éléments de la liste.

```
PROGRAMME Parcours_liste_entier
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_entier
FINSTRUCTURE

STRUCTURE liste_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

VAR
   ma_liste : liste_entier
   *maillon_courant <- NULL : maillon_entier
DEBUT
   maillon_courant <- ma_liste.premier
   // Le dernier maillon pointe sur NULL
   TANTQUE maillon_courant ≠ NULL
```

```
    FAIRE
        ECRIRE("Valeur : ", maillon_courant->valeur)
        maillon_courant <- maillon_courant->suivant
    FINTANTQUE
FIN
```

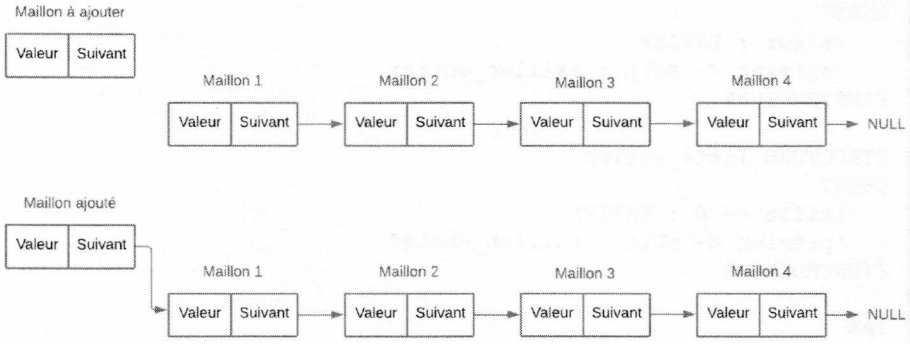
La difficulté de ce parcours réside en l'utilisation d'un pointeur de maillon qui prendra comme valeurs les maillons de la liste, les uns à la suite des autres.

2.1.3 Ajout d'un élément

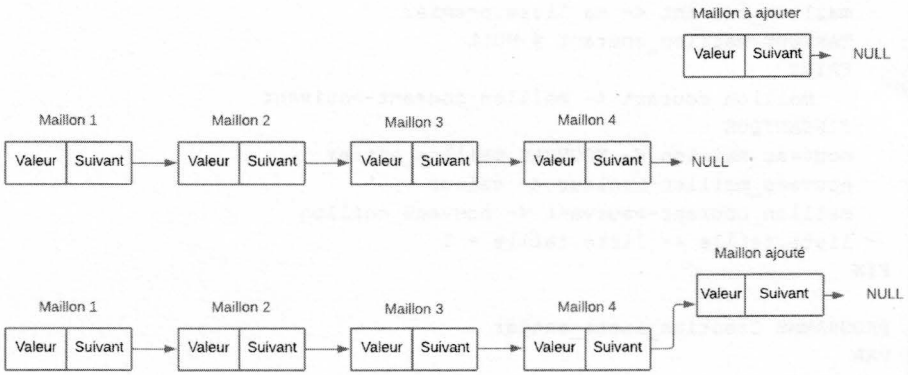
Nous pouvons ajouter un maillon à une liste, soit comme premier élément, soit comme dernier :

- La procédure `ajouter_debut` prend en paramètres une liste et une valeur, et ajoute cette valeur comme premier élément de la liste. Sa logique est illustrée dans la figure suivante.
- La procédure `ajouter_fin` prend en paramètres une liste et une valeur, et ajoute cette valeur comme dernier élément de la liste. Son comportement est représenté plus loin dans la section.

Créons maintenant un algorithme pour créer et ajouter quelques valeurs à une liste d'entiers.



Ajout d'un élément en tête de liste



Ajout d'un maillon en fin de la liste

```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_entier
FINSTRUCTURE

STRUCTURE liste_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

PROCEDURE ajouter_debut(E/S: liste : liste_entier, E : valeur : ENTIER)
VAR
   *maillon : maillon_entier
DEBUT
   maillon <- NOUVEAU maillon_entier
   maillon->valeur <- valeur
   maillon->suivant <- liste.premier
   liste.premier <- maillon
   liste.taille <- liste.taille + 1
FIN

PROCEDURE ajouter_fin(E/S: liste : liste_entier, E : valeur : ENTIER)
VAR
   *maillon_courant <- NULL : maillon_entier
   *nouveau_maillon <- NULL : maillon_entier
DEBUT
```

```
   maillon_courant <- ma_liste.premier
   TANTQUE maillon_courant ≠ NULL
   FAIRE
      maillon_courant <- maillon_courant->suivant
   FINTANTQUE
   nouveau_maillon <- NOUVEAU maillon_entier
   nouveau_maillon->valeur <- valeur
   maillon_courant->suivant <- nouveau_maillon
   liste.taille <- liste.taille + 1
FIN

PROGRAMME Creation_liste_entier
VAR
   ma_liste : liste_entier
   nouvelle_valeur, i : ENTIER
   *maillon_courant <- NULL : maillon_entier
DEBUT
   POUR i ALLANT DE 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez la valeur ", i)
      valeur <- LIRE()
      ajout_debut(ma_liste, valeur)
   FINPOUR
   POUR i ALLANT DE 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez la valeur ", i)
      valeur <- LIRE()
      ajout_fin(ma_liste, valeur)
   FINPOUR
   // Affichons notre liste en supposant que l'utilisateur
a entré les entiers de 1 à 10 dans l'ordre croissant
   maillon_courant <- ma_liste.premier
   TANTQUE maillon_courant ≠ NULL
   FAIRE
      ECRIRE("Valeur : ", maillon_courant->valeur)
      maillon_courant <- maillon_courant->suivant
   FINTANTQUE
   // valeur de la liste : 5 4 3 2 1 6 7 8 9 10
FIN
```

Pour ajouter au début, nous créons un nouveau pointeur de maillon qui prend comme `suivant` les éléments de la liste avec une affectation au premier élément de la liste. Il nous suffit ensuite de remplacer le premier maillon de la liste par ce nouveau maillon.

Pour ajouter à la fin, nous devons parcourir la liste jusqu'au dernier élément et le faire pointer sur le nouveau maillon et non plus `NULL`. Par défaut, le pointeur suivant du maillon inséré a comme valeur `NULL`, il est donc le dernier élément de la liste.

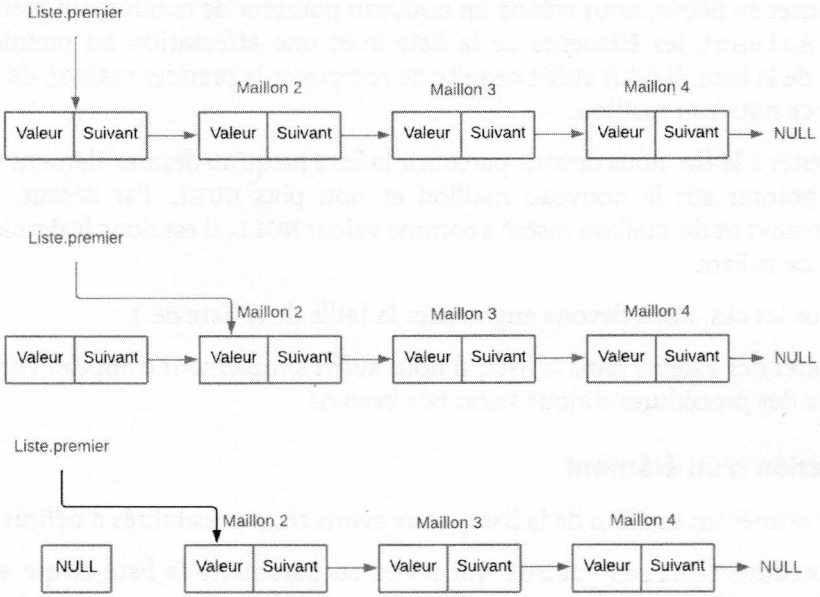
Dans tous les cas, nous devons augmenter la taille de la liste de 1.

Pour insérer des valeurs dans la liste, il nous suffit simplement d'appeler l'une ou l'autre des procédures d'ajout selon nos besoins.

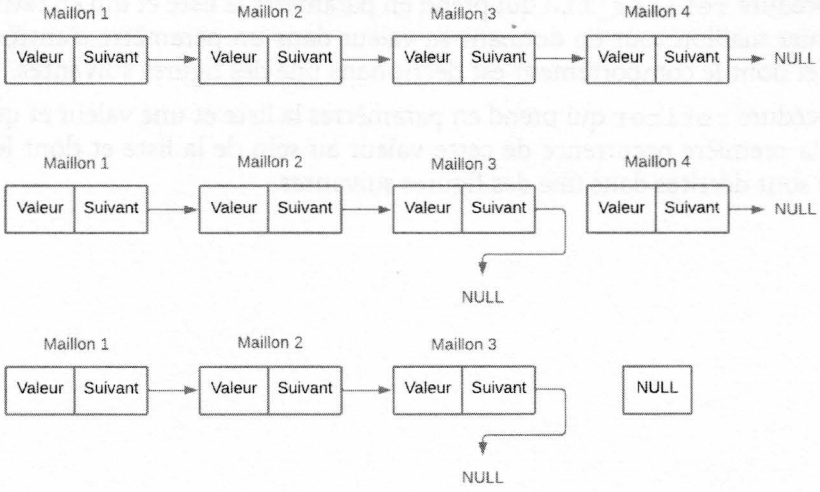
2.1.4 Suppression d'un élément

Pour supprimer un maillon de la liste, nous avons trois possibilités à définir :

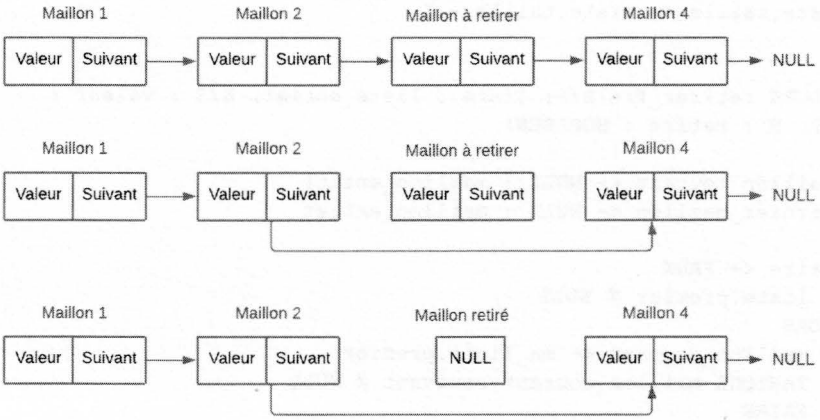
- La procédure `retirer_debut` qui prend en paramètre la liste et qui en retire le premier maillon tout en donnant sa valeur dans un paramètre d'entrée/sortie et dont la logique est illustrée dans la figure suivante.
- La procédure `retirer_fin` qui prend en paramètre la liste et qui en retire le dernier maillon tout en donnant sa valeur dans un paramètre d'entrée/sortie et dont le comportement est décrit dans une des figures suivantes.
- La procédure `retirer` qui prend en paramètres la liste et une valeur et qui retire la première occurrence de cette valeur au sein de la liste et dont les étapes sont décrites dans une des figures suivantes.



Suppression du premier élément d'une liste



Suppression du dernier élément d'une liste



Retirer un maillon au milieu d'une liste

```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_entier
FINSTRUCTURE

STRUCTURE liste_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

PROCEDURE retirer_debut(E/S: liste : liste_entier, E/S : valeur :
ENTIER, S : retire : BOOLEEN)
VAR
   *maillon : maillon_entier
DEBUT
   retire <- FAUX
   SI liste.premier ≠ NULL
   ALORS
      valeur <- liste.premier->valeur
      maillon <- liste.premier
      liste.premier <- liste.premier->suivant
      maillon <- NULL
      retire <- VRAI
   FINSI
```

```
   liste.taille <- liste.taille - 1
FIN

PROCEDURE retirer_fin(E/S: liste : liste_entier, E/S : valeur :
ENTIER, S : retire : BOOLEEN)
VAR
   *maillon_courant <- NULL : maillon_entier
   *dernier_maillon <- NULL : maillon_entier
DEBUT
   retire <- FAUX
   SI liste.premier ≠ NULL
   ALORS
      maillon_courant <- ma_liste.premier
      TANTQUE maillon_courant->suivant ≠ NULL
      FAIRE
         dernier_maillon <- maillon_courant
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      liste.taille <- liste.taille - 1
      valeur <- maillon_courant->valeur
      maillon_courant <- NULL
      dernier_maillon.suivant <- NULL
      retire <- VRAI
   FINSI
FIN

PROCEDURE retirer(E/S: liste : liste_entier, E : valeur : ENTIER,
S : retire : BOOLEEN)
VAR
   *maillon_courant <- NULL : maillon_entier
   *maillon_avant <- NULL : maillon_entier
DEBUT
   retire <- FAUX
   SI liste.premier ≠ NULL
   ALORS
      maillon_courant <- ma_liste.premier
      TANTQUE maillon_courant->valeur ≠ valeur ET maillon_courant
? NULL
      FAIRE
         maillon_avant <- maillon_courant
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      SI maillon_courant ≠ NULL
      ALORS
```

```
            maillon_avant->suivant <- maillon_courant->suivant
            maillon_courant <- NULL
            liste.taille <- liste.taille - 1
            retire <- VRAI
         FINSI
      FINSI
FIN

PROGRAMME Creation_suppression_maillon_liste_entier

VAR
   ma_liste : liste_entier
   nouvelle_valeur, i : ENTIER
   *maillon_courant <- NULL : maillon_entier
   ok : BOOLEEN
DEBUT
   // Création de la liste
   POUR i ALLANT DE 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez la valeur ", i)
      valeur <- LIRE()
      ajout_debut(ma_liste, valeur)
   FINPOUR
   POUR i ALLANT DE 1 A 5 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez la valeur ", i)
      valeur <- LIRE()
      ajout_fin(ma_liste, valeur)
   FINPOUR
   // Affichons notre liste en supposant que l'utilisateur a entré
les entiers de 1 à 10 dans l'ordre croissant
   maillon_courant <- ma_liste.premier
   TANTQUE maillon_courant ≠ NULL
   FAIRE
      ECRIRE("Valeur : ", maillon_courant->valeur)
      maillon_courant <- maillon_courant->suivant
   FINTANTQUE
   // valeur de la liste : 5 4 3 2 1 6 7 8 9 10
   // Suppression de trois éléments de la liste
   retirer_debut(ma_liste, nouvelle_valeur, ok)
   SI ok
      ECRIRE("La valeur retirée est : ", nouvelle_valeur)
   ALORS
   SINON
```

```
      ECRIRE("Nous ne pouvons retirer une valeur d'une liste vide")
   FINSI
   retirer_fin(ma_liste, nouvelle_valeur, ok)
   SI ok
      ECRIRE("La valeur retirée est : ", nouvelle_valeur)
   ALORS
   SINON
      ECRIRE("Nous ne pouvons retirer une valeur d'une liste vide")
   FINSI
   retirer (ma_liste, 1, ok)
   SI ok
      ECRIRE("La valeur retirée est : ", nouvelle_valeur)
   ALORS
   SINON
      ECRIRE("Nous ne pouvons retirer une valeur d'une liste vide
ou la valeur n'est pas dans la liste")
   FINSI
   // valeur de la liste :  4 3 2 5 6 7 8 9
FIN
```

Dans les procédures retirant des éléments, nous avons ajouté un booléen en paramètre de sortie pour indiquer que nous avons effectivement retirer un maillon de la chaîne. Cela nous permet de gérer facilement le cas des listes vides ainsi que le fait que la valeur n'existe pas dans la liste.

Pour chaque type de suppression de maillon, il est **primordial de désallouer la mémoire du maillon** à retirer en l'affectant à la valeur `NULL`. Dans le cas contraire, nous créons une fuite de mémoire, donc un problème majeur pour la gestion de la mémoire du programme par l'ordinateur.

Pour retirer le premier maillon, nous affectons le premier maillon de la liste à son suivant tout en le désallouant par la suite.

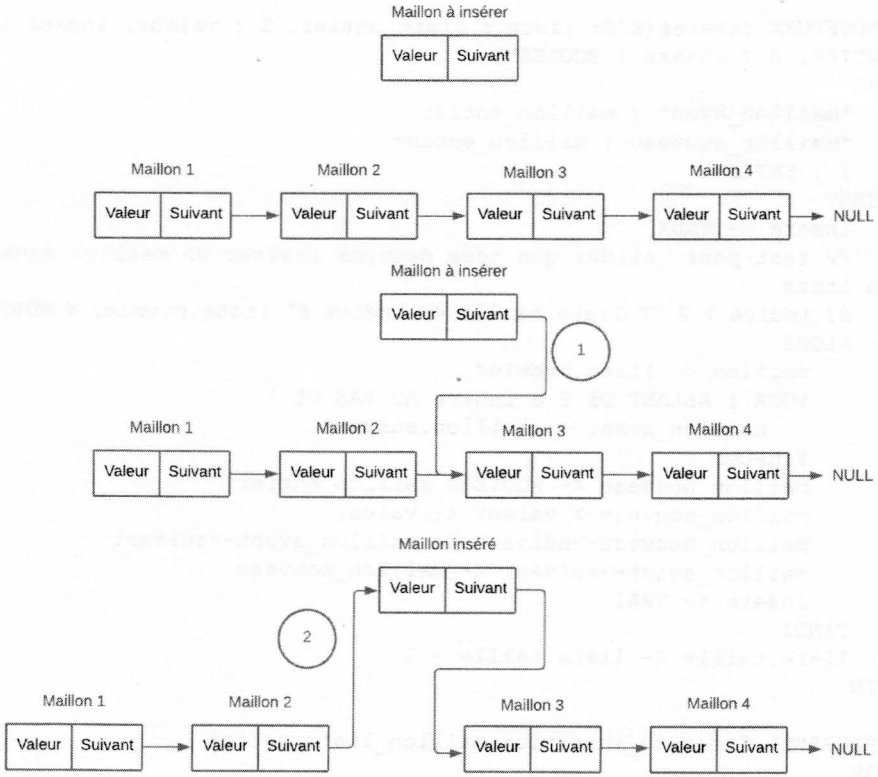
Pour retirer le dernier maillon, nous parcourons la liste jusqu'à l'avant-dernier maillon. Nous désallouons le pointeur `suivant` de maillon pour effacer le dernier élément de la liste.

Pour retirer un maillon au milieu de la liste, nous recherchons la valeur du maillon dans un parcours de la liste. Une fois cette valeur trouvée, nous faisons "sauter" un maillon au maillon précédent tout en désallouant le maillon à retirer.

2.1.5 Insertion d'un élément

Pour insérer un maillon dans une liste, nous allons écrire une procédure `insérer` qui prend en paramètre la liste, la valeur à insérer ainsi que son indice. C'est notamment pour cette procédure que la liste possède un champ représentant sa taille.

Cette procédure insérera la valeur dans un nouveau maillon à la position demandée. Le comportement de cette procédure est représenté dans la figure ci-après.



Insertion d'un maillon dans une liste

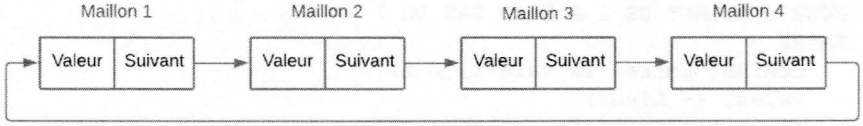
```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_entier
FINSTRUCTURE

STRUCTURE liste_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

PROCEDURE inserer(E/S: liste : liste_entier, E : valeur, indice :
ENTIER, S : insere : BOOLEEN)
VAR
   *maillon_avant : maillon_entier
   *maillon_nouveau : maillon_entier
   i : ENTIER
DEBUT
   insere <- FAUX
   // test pour valider que nous pouvons insérer un maillon dans
la liste
   SI indice > 0 ET liste.taille <= indice ET liste.premier ≠ NULL
   ALORS
      maillon <- liste.premier
      POUR i ALLANT DE 2 à indice AU PAS DE 1
         maillon_avant <- maillon.suivant
      FINPOUR
      maillon_nouveau <- NOUVEAU maillon_entier
      maillon_nouveau-> valeur <- valeur
      maillon_nouveau->suivant <- maillon_avant->suivant
      maillon_avant->suivant <- maillon_nouveau
      insere <- VRAI
   FINSI
   liste.taille <- liste.taille - 1
FIN

PROGRAMME Creation_insertion_maillon_liste_entier
VAR
   ma_liste : liste_entier
   nouvelle_valeur, i : ENTIER
   *maillon_courant <- NULL : maillon_entier
   ok : BOOLEEN
DEBUT
```

```
    // Création de la liste
    POUR i ALLANT DE 1 A 5 AU PAS DE 1
    FAIRE
       ECRIRE("Entrez la valeur ", i)
       valeur <- LIRE()
       ajout_debut(ma_liste, valeur)
    FINPOUR
    POUR i ALLANT DE 1 A 5 AU PAS DE 1
    FAIRE
       ECRIRE("Entrez la valeur ", i)
       valeur <- LIRE()
       ajout_fin(ma_liste, valeur)
    FINPOUR
    // Affichons notre liste en supposant que l'utilisateur a entré
les entiers de 1 à 10 dans l'ordre croissant
    maillon_courant <- ma_liste.premier
    TANTQUE maillon_courant ≠ NULL
    FAIRE
       ECRIRE("Valeur : ", maillon_courant->valeur)
    FINTANTQUE
    // valeur de la liste : 5 4 3 2 1 6 7 8 9 10
    // Insertion de la valeur 42 à l'indice 3
    inserer(ma_liste, 42, 3, ok)
    SI ok
       ECRIRE("Insertion réussie")
    SINON
       ECRIRE("Indice trop grand ou trop petit ou liste vide")
    // valeur de la liste :  4 3 42 2 1 6 7 8 9 10
FIN
```

La logique d'insertion d'une valeur dans une liste est la suivante : après avoir vérifié que l'insertion est possible (indice positif et inclus dans la taille de la liste qui est non vide), nous parcourons la liste jusqu'au maillon d'indice voulu que nous appelons `maillon_avant`. Une fois ce maillon récupéré, nous créons le nouveau maillon avec son champ `suivant` qui pointera sur le `suivant` du `maillon_avant`, puis le `suivant` de `maillon_avant` sur ce nouveau maillon.

2.2 Listes chaînées circulaires

Une liste chaînée circulaire est une liste simplement chaînée dont le champ suivant du dernier maillon pointe sur le premier maillon de la liste, comme illustré dans la figure ci-dessous.



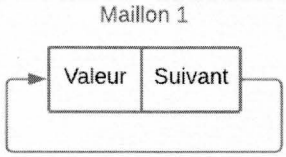
Liste chaînée circulaire

2.2.1 Création et parcours

Au niveau des `STRUCTURE`, pour représenter une liste circulaire, nous pouvons utiliser celles définies pour la liste simplement chaînée :

```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon
FINSTRUCTURE

STRUCTURE liste_circulaire_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE
```

La seule différence avec la liste simplement chaînée réside en la gestion du champ `suivant` du dernier maillon : il ne pointe pas sur `NULL` mais sur le premier maillon. Ainsi, une liste d'un seul élément aura son maillon qui pointera sur lui-même comme représenté dans la figure ci-dessous.



Liste circulaire d'un seul maillon

Commençons par créer une liste d'un maillon que nous allons parcourir par la suite :

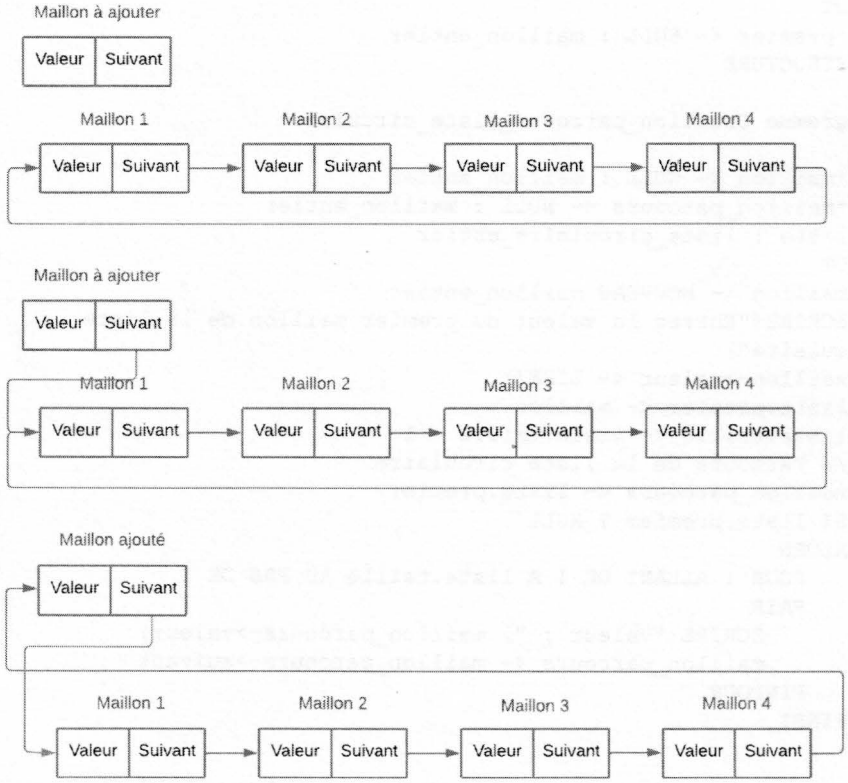
```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon

STRUCTURE liste_circulaire_entier
DEBUT
   *premier <- NULL : maillon_entier
FINSTRUCTURE

Programme Creation_parcours_liste_circulaire
VAR
   *maillon <- NULL : maillon_entier
   *maillon_parcours <- NULL : maillon_entier
   liste : liste_circulaire_entier
DEBUT
   maillon <- NOUVEAU maillon_entier
   ECRIRE("Entrez la valeur du premier maillon de la liste
circulaire")
   maillon->valeur <- LIRE()
   liste.premier <- maillon
   liste.taille <- liste.taille + 1
   // Parcours de la liste circulaire
   maillon_parcours <- liste.premier
   SI liste.premier ? NULL
   ALORS
      POUR i ALLANT DE 1 A liste.taille AU PAS DE 1
      FAIR
         ECRIRE("Valeur : ", maillon_parcours->valeur)
         maillon_parcours <- maillon_parcours->suivant
      FINPOUR
   FINSI
FIN
```

Pour parcourir une liste simplement chaînée circulaire, nous ne pouvons utiliser la structure itérative TANTQUE identique à celle du parcours d'une liste simplement chaînée : cela entraînerait une boucle infinie affichant les valeurs de la liste sans jamais s'arrêter. Nous sommes donc obligés d'utiliser une structure itérative POUR qui va itérer sur les positions des maillons et s'arrêter au dernier maillon de la liste, et, ce grâce au champ taille de la liste.

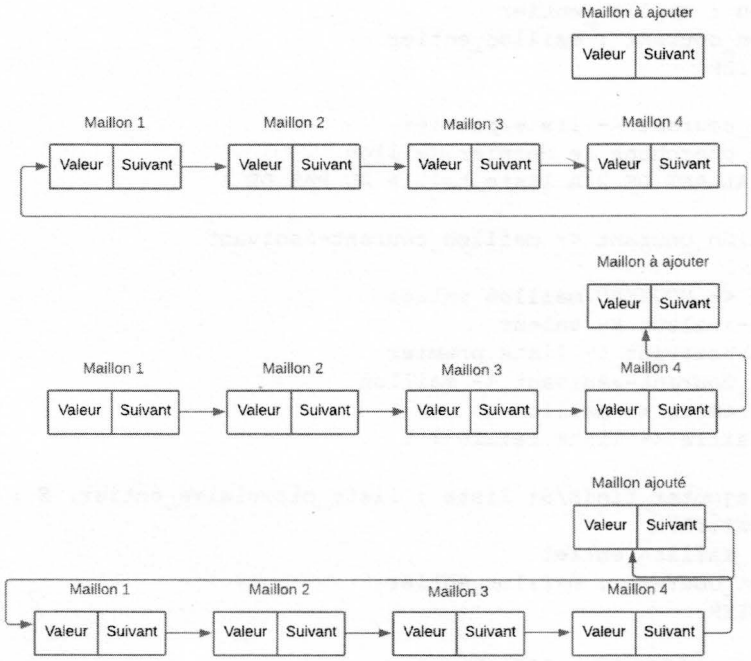
2.2.2 Ajout d'un élément

Les deux procédures d'ajout en début ou en fin de liste de la liste simplement chaînée doivent être **adaptées** pour la liste circulaire pour la gestion du champ `suivant` du dernier maillon qui pointe sur le premier maillon de la liste.

La logique de la procédure `ajouter_debut` est représentée dans la figure suivante, et celle de la procédure `ajouter_fin` dans la figure d'après.



Ajout en début d'une liste circulaire



Ajout en fin d'une liste circulaire

Construisons maintenant les algorithmes de ces deux procédures. Pour des raisons de simplicité, nous estimons qu'un ajout ne se fera jamais sur une liste circulaire vide.

```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon
FINSTRUCTURE

STRUCTURE liste_circulaire_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE
PROCEDURE ajouter_debut(E/S: liste : liste_circulaire_entier, S :
valeur : ENTIER)
VAR
```

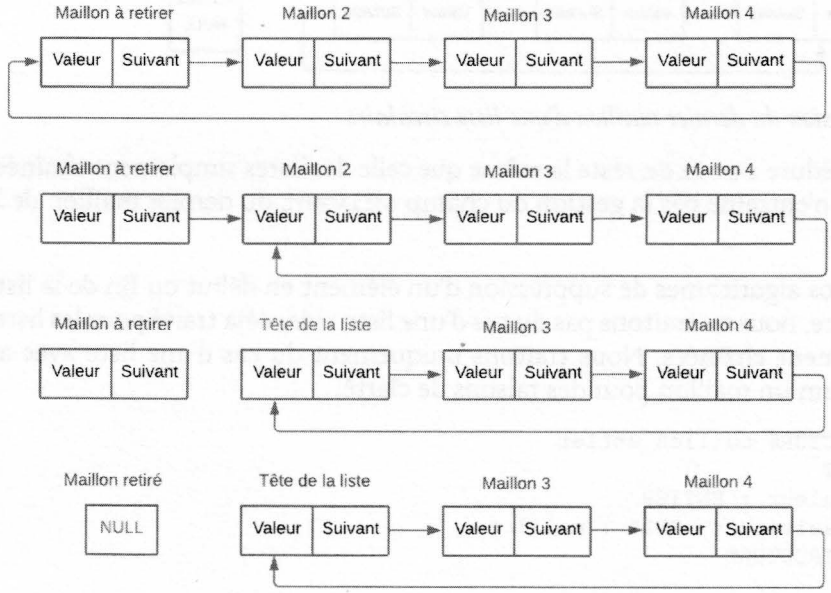
```
   *maillon : maillon_entier
   *maillon_courant : maillon_entier
   i : ENTIER
DEBUT
   maillon_courant <- liste.premier
   // Nous cherchons le dernier maillon
   POUR i ALLANT DE 2 A liste.taille AU PAS DE 1
   FAIRE
      maillon_courant <- maillon_courant->suivant
   FINPOUR
   maillon <- NOUVEAU maillon_entier
   maillon->valeur <- valeur
   maillon->suivant <- liste.premier
   maillon_courant->suivant <- maillon
   liste.premier <- maillon
   liste.taille <- liste.taille + 1
FIN
PROCEDURE ajouter_fin(E/S: liste : liste_circulaire_entier, S :
valeur : ENTIER)
*maillon : maillon_entier
   *maillon_courant : maillon_entier
   i : ENTIER
DEBUT
   maillon <- NOUVEAU maillon_entier
   maillon->valeur <- valeur
   maillon->suivant <- liste.premier
   liste.premier <- maillon
   maillon_courant <- liste.premier
   POUR i ALLANT DE 2 A liste.taille AU PAS DE 1
   FAIRE
      maillon_courant <- maillon_courant->suivant
   FINPOUR
   maillon <- NOUVEAU maillon_entier
   maillon->valeur <- valeur
   maillon->suivant <- liste.premier
   maillon_courant->suivant <- maillon
   liste.taille <- liste.taille + 1
FIN
```

Pour ajouter, il nous faut toujours parcourir la liste afin de récupérer le dernier maillon et de changer son pointeur `suivant` sur le bon maillon.

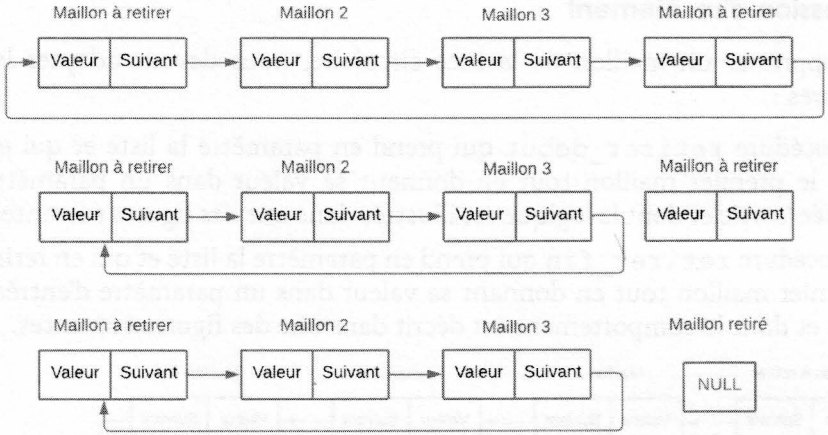
2.2.3 Suppression d'un élément

Pour supprimer un maillon de la liste circulaire, nous devons adapter les procédures :

- La procédure `retirer_debut` qui prend en paramètre la liste et qui en retire le premier maillon tout en donnant sa valeur dans un paramètre d'entrée/sortie et dont la logique est illustrée dans une des figures suivantes;
- La procédure `retirer_fin` qui prend en paramètre la liste et qui en retire le dernier maillon tout en donnant sa valeur dans un paramètre d'entrée/sortie et dont le comportement est décrit dans une des figures suivantes.



Suppression du premier maillon d'une liste circulaire



Suppression du dernier maillon d'une liste circulaire

La procédure `retirer` reste la même que celle des listes simplement chaînées car elle n'entraîne pas la gestion du champ `suivant` du dernier maillon de la liste.

Dans nos algorithmes de suppression d'un élément en début ou fin de la liste circulaire, nous ne traitons pas du cas d'une liste vide, déjà traité pour les listes simplement chaînées. Nous traitons uniquement du cas d'une liste avec au minimum un maillon pour des raisons de clarté.

```
STRUCTURE maillon_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon
FINSTRUCTURE

STRUCTURE liste_circulaire_entier
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

PROCEDURE retirer_debut(E/S: liste   liste_circulaire_entier,
E/S: valeur : ENTIER, S : retire : BOOLEEN)
VAR
   *maillon : maillon_entier
```

```
   *maillon_courant : maillon_entier
   i : ENTIER
DEBUT
   retire <- FAUX
   SI liste.taille = 1
   ALORS
      liste.premier <- NULL
   SINON
         maillon <- liste.premier
      valeur <- maillon->valeur
      maillon_courant <- liste.premier
      POUR i ALLANT DE 2 A liste.taille AU PAS DE 1
      FAIRE
         maillon_courant <- maillon_courant->suivant
      FINPOUR
      maillon_courant->suivant <- maillon->suivant
      liste.premier <- maillon->suivant
      valeur <- maillon->valeur
      maillon <- NULL
      liste.taille <- liste.taille - 1
      retire <- VRAI
   FINSI
FIN

PROCEDURE retirer_fin(E/S: liste : liste_circulaire_entier,
E/S : valeur : ENTIER, S : retire : BOOLEEN)
VAR
   *maillon : maillon_entier
   *maillon_courant : maillon_entier
   i : ENTIER
DEBUT
   retire <- FAUX
   SI liste.taille = 1
   ALORS
      liste.premier <- NULL
   SINON

      maillon_courant <- liste.premier
      POUR i ALLANT DE 2 A liste.taille AU PAS DE 1
      FAIRE
         maillon <- maillon_courant
         maillon_courant <- maillon_courant->suivant
      FINPOUR
      valeur <- maillon_courant->valeur
```

```
        maillon_courant <- NULL
        maillon->suivant <- liste.premier
        liste.taille <- liste.taille - 1
    FINSI
FIN
```

Le cas particulier de la suppression d'un maillon en fin ou début d'une liste circulaire avec un seul élément est résolu facilement : le maillon premier de la liste passe à `NULL` pour avoir une liste vide.

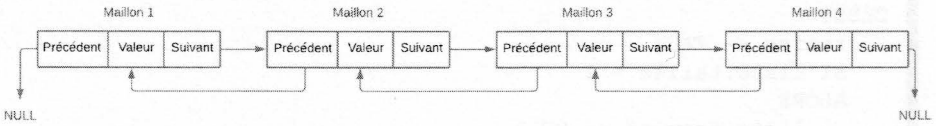
2.2.4 Insérer un élément

Pour insérer un nouveau maillon au sein d'une liste circulaire, nous pouvons utiliser la même procédure d'insertion que celle définie pour la liste simplement chaînée car l'insertion n'affecte pas normalement le dernier maillon de la chaîne (vous devez utiliser la procédure `ajout_dernier` pour cela).

Corsons davantage nos listes avec un lien double entre leurs maillons.

2.3 Listes doublement chaînées

Une liste **doublement chaînée** est une liste où chaque maillon possède un pointeur vers le maillon **précédent**, en plus du pointeur sur son **successeur**. Comme le montre la figure ci-dessous, le pointeur `précédent` du premier maillon a comme valeur `NULL` comme le pointeur `suivant` du dernier maillon de la liste.



Liste doublement chaînée

2.3.1 Création et parcours

Nous devons adapter la STRUCTURE maillons des listes simplement chaînées pour les listes doublement chaînées en ajoutant un nouveau champ pour pointer sur le maillon précédent :

```
STRUCTURE maillon_double_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_double_entier
   *precedent <- NULL : maillon_double_entier
FINSTRUCTURE

STRUCTURE liste_doublement_chainee
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE
```

Nous allons maintenant écrire un algorithme qui crée une liste doublement chaînée d'un élément et qui la parcourt :.

```
PROGRAMME Parcours_creation_liste_doublement_chainee
STRUCTURE maillon_double_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_double_entier
   *precedent <- NULL : maillon_double_entier
FINSTRUCTURE

STRUCTURE liste_doublement_chainee
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

VAR
   *maillon <- NULL : maillon_entier
   *maillon_parcours <- NULL : maillon_entier
   liste : liste_doublement_chainee
   i : ENTIER
DEBUT
   maillon <- NOUVEAU maillon_entier
   ECRIRE("Entrez la valeur du premier maillon de la liste
```

```
circulaire")
   maillon->valeur <- LIRE()
   liste.premier <- maillon
   liste.taille <- liste.taille + 1
   // Parcours de la liste doublement chaînée
   maillon_courant <- ma_liste.premier
   POUR i ALLANT DE 1 A liste.taille AU PAS DE 1
   FAIRE
      ECRIRE("Valeur : ", maillon_courant->valeur)
      maillon_courant <- maillon_courant->suivant
   FINPOUR
FIN
```

Nous remarquons que rien ne change sur la logique de création ni du parcours d'une liste doublement chaînée en comparaison avec la simplement chaînée.

Les difficultés commenceront avec l'ajout ou la suppression d'un élément de la liste doublement chaînée.

2.3.2 Ajout et insertion d'un élément

Pour augmenter la taille d'une liste doublement chaînée avec un nouveau maillon, nous devons adapter une nouvelle fois les procédures d'ajout en début et en fin et la procédure d'insertion pour ajouter la gestion du pointeur sur le maillon précédent :

```
STRUCTURE maillon_double_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_double_entier
   *precedent <- NULL : maillon_double_entier
FINSTRUCTURE

STRUCTURE liste_doublement_chainee
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

PROCEDURE ajouter_debut(E/S: liste : liste_doublement_chainee,
S : valeur : ENTIER)
VAR
   *maillon : maillon_double_entier
```

```
DEBUT
   SI liste.premier = NULL
   ALORS
      maillon <- NOUVEAU maillon_entier
      maillon->valeur <- valeur
      liste.premier <- maillon
   SINON
      maillon <- NOUVEAU maillon_entier
      maillon->valeur <- valeur
      maillon->suivant <- liste.premier
      liste.premier->precedent <- maillon
      liste.premier <- maillon
      liste.taille <- liste.taille + 1
   FINSI
FIN

PROCEDURE ajouter_fin(E/S: liste : liste_doublement_chainee,
S : valeur : ENTIER)
VAR
   *maillon_courant <- NULL : maillon_double_entier
   *nouveau_maillon <- NULL : maillon_double_entier
DEBUT
   SI liste.premier = NULL
   ALORS
   ajouter_debut(liste, valeur)
   SINON
      maillon_courant <- ma_liste.premier
      TANTQUE maillon_courant ? NULL
      FAIRE
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      nouveau_maillon <- NOUVEAU maillon_entier
      nouveau_maillon->valeur <- valeur
      maillon_courant->suivant <- nouveau_maillon
      nouveau_maillon->précédent <- maillon_courant
      liste.taille <- liste.taille + 1
FIN

PROCEDURE inserer(E/S: liste : liste_entier, E : valeur, indice :
ENTIER, S : insere : BOOLEEN)
VAR
   *maillon_avant : maillon_entier
   *maillon_nouveau : maillon_entier
   I : ENTIER
```

```
DEBUT
   insere <- FAUX
   SI indice > 0 ET liste.taille < indice - 1 ET liste.premier ≠ NULL
   ALORS
      maillon <- liste.premier
      POUR i ALLANT DE 2 à indice - 1 AU PAS DE 1
      FAIRE
         maillon_avant <- maillon.suivant
      FINPOUR
      maillon_nouveau <- NOUVEAU maillon_entier
      maillon_nouveau-> valeur <- valeur
      maillon_nouveau->precedent <- maillon_avant
      maillon_nouveau->suivant <- maillon_avant->suivant
      maillon_avant->suivant <- maillon_nouveau
      insere <- VRAI
   FINSI
   liste.taille <- liste.taille 1 1
FIN
```

Nous remarquons que nous avons juste ajouté la gestion du pointeur précédent pour gérer les ajouts dans une liste doublement chaînée.

2.3.3 Suppression d'un élément

L'adaptation des procédures de suppression d'un maillon dans une liste doublement chaînée est la même que celle de l'ajout : nous devons juste ajouter la gestion du champ précédent.

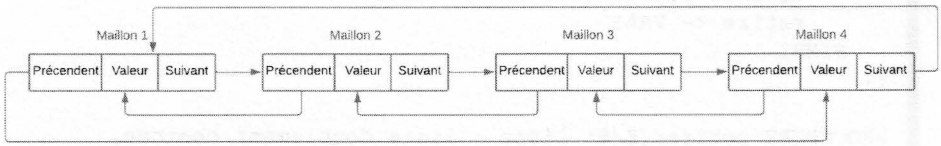
```
STRUCTURE maillon_double_entier
DEBUT
   valeur : ENTIER
   *suivant <- NULL : maillon_double_entier
   *precedent <- NULL : maillon_double_entier
FINSTRUCTURE

STRUCTURE liste_doublement_chainee
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon_entier
FINSTRUCTURE

PROCEDURE retirer_debut(E/S: liste : liste_doublement_chainee,
E/S : valeur : ENTIER, S : retire : BOOLEEN)
VAR
```

```
   *maillon : maillon_double_entier
DEBUT
   retire <- FAUX
   SI liste.premier ≠ NULL
   ALORS
      valeur <- liste.premier->valeur
      maillon <- liste.premier
      liste.premier <- liste.premier->suivant
      liste.precedent <- NULL
      maillon <- NULL
      retire <- VRAI
   FINSI
   liste.taille <- liste.taille 1 1
FIN

PROCEDURE retirer_fin(E/S: liste : liste_doublement_chainee,
E/S : valeur : ENTIER, S : retire : BOOLEEN)
VAR
   *maillon_courant <- NULL : maillon_double_entier
   *dernier_maillon <- NULL : maillon_double_entier
DEBUT
   retire <- FAUX
   SI liste.premier ≠ NULL
   ALORS
      maillon_courant <- ma_liste.premier
      TANTQUE maillon_courant->suivant ≠ NULL
      FAIRE
         dernier_maillon <- maillon_courant
         maillon_courant <- maillon_courant->suivant
         FINTANTQUE
         liste.taille <- liste.taille - 1
      valeur <- maillon_courant->valeur
      dernier_maillon.precedent <- maillon_courant->precedent
      maillon_courant <- NULL
      dernier_maillon.suivant <- NULL
      retire <- VRAI
   FINSI
FIN

PROCEDURE retirer(E/S: liste : liste_doublement_chainee,
E : valeur : ENTIER, S : retire : BOOLEEN)
VAR
   *maillon_courant <- NULL : maillon_double_entier
   *maillon_avant <- NULL : maillon_double_entier
```

```
DEBUT
   retire <- FAUX
   SI liste.premier ≠ NULL
   ALORS
      maillon_courant <- ma_liste.premier
      TANTQUE maillon_courant->valeur ≠ valeur ET maillon_courant
≠ NULL
      FAIRE
         maillon_avant <- maillon_courant
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      SI maillon_courant ≠ NULL
      ALORS
         maillon_avant->suivant <- maillon_courant->suivant
         maillon_avant->precedent <- maillon_courant->precedent
         maillon_courant <- NULL
         liste.taille <- liste.taille - 1
         retire <- VRAI

      FINSI
      liste.taille <- liste.taille - 1
      valeur <- maillon_courant->valeur
      maillon_courant <- NULL
      dernier_maillon.suivant <- NULL
      retire <- VRAI
   FINSI
FIN
```

Remarque

Nous n'avons parlé que de la liste simplement chaînée circulaire. Il existe également la liste doublement chaînée circulaire, une liste hybride entre notre liste circulaire et la liste doublement chaînée vue précédemment. La représentation d'un tel type de liste est illustré dans la figure ci-après :



Liste doublement chaînée circulaire

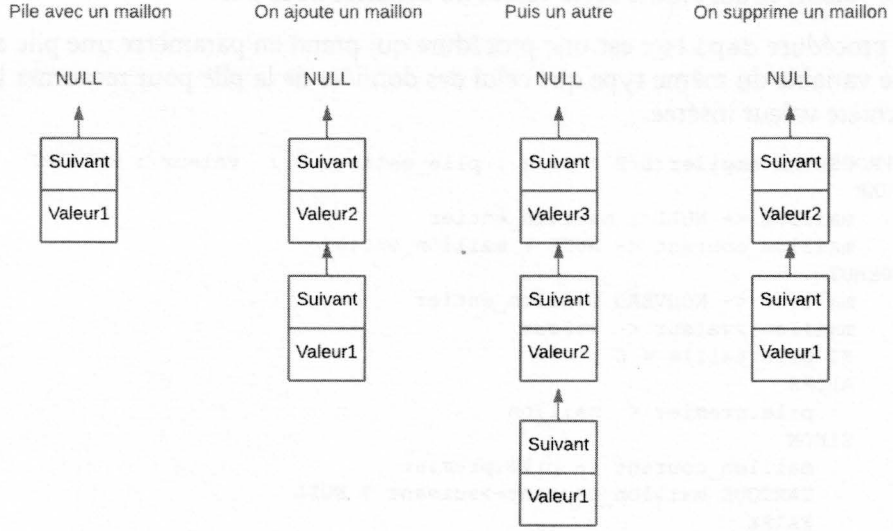
2.4 Piles et Files

2.4.1 Piles ou LIFO

Une **Pile** est une liste simplement chaînée particulière : **nous ne pouvons ajouter ou retirer que le dernier élément**.

Comme dans une pile de vêtements que nous pouvons créer, le vêtement que nous pouvons prendre est le dernier posé, le premier vêtement ne pouvant être récupéré que si tous les autres vêtements ont été retirés de la pile.

Cette contrainte explique son acronyme **LIFO** pour *Last In First Out*, le dernier maillon est le seul à pouvoir être retiré et le premier maillon est toujours le plus ancien dans l'ordre d'insertion, comme l'illustre la figure ci-dessous.



Fonctionnement d'une pile

De ce fait, nous n'avons qu'une seule procédure pour ajouter un maillon et une seule procédure pour retirer un maillon avec les piles, qui deviendront respectivement les procédures `empiler` et `depiler`. De plus, les procédures `retirer` et `inserer` des listes ne peuvent pas être implémentées sur les piles du fait de leur contrainte de gestions des éléments.

Deux exemples d'utilisations de piles par l'ordinateur sont la gestion des registres des processeurs en bas niveau, et en haut niveau la mémorisation de l'historique d'un navigateur Internet par exemple.

Pour définir une liste, nous pouvons donc utiliser notre `STRUCTURE` maillon, ou `maillon_entier` pour une pile d'entiers. La `STRUCTURE` de la pile est également semblable à celle de la liste simplement chaînée.

```
STRUCTURE pile_entier
DEBUT
   *premier <- NULL : maillon_entier
   taille <- 0 : ENTIER
FINSTRUCTURE
```

La procédure `empiler` est une procédure qui prend en paramètres une pile et une valeur, et qui ajoute cette valeur au sommet de la pile.

La procédure `depiler` est une procédure qui prend en paramètre une pile et une variable du même type que celui des données de la pile pour retourner la dernière valeur insérée.

```
PROCEDURE empiler(E/S : pile : pile_entier, E :  valeur : ENTIER)
VAR
   maillon <- NULL : maillon_entier
   maillon_courant <- NULL : maillon_entier
DEBUT
   maillon <- NOUVEAU maillon_entier
   maillon->valeur <- valeur
   SI pile.taille = 0
   ALORS
      pile.premier <- maillon
   SINON
      maillon_courant <- pile.premier
      TANTQUE maillon_courant->suivant ? NULL
      FAIRE
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      maillon_courant->suivant <- maillon
   FINSI
   pile.taille <- pile.taille + 1
FIN

PROCEDURE depiler(E/S : pile : pile_entier, E/S : valeur : ENTIER)
VAR
```

```
    maillon_courant <- NULL : maillon_entier
DEBUT
   SI pile.taille > 0
   ALORS
      maillon_courant <- pile.premier
      TANTQUE maillon_courant->suivant ? NULL
      FAIRE
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      valeur <- maillon_courant->valeur
      maillon_courant->NULL
      pile.taille <- pile.taille - 1
   FINSI
FIN
```

Nous pouvons remarquer que la procédure `empiler` correspond à la procédure `ajouter_fin` de la liste simplement chaînée. La procédure `depiler` quant à elle correspond à la procédure `retirer_fin` de la liste simplement chaînée.

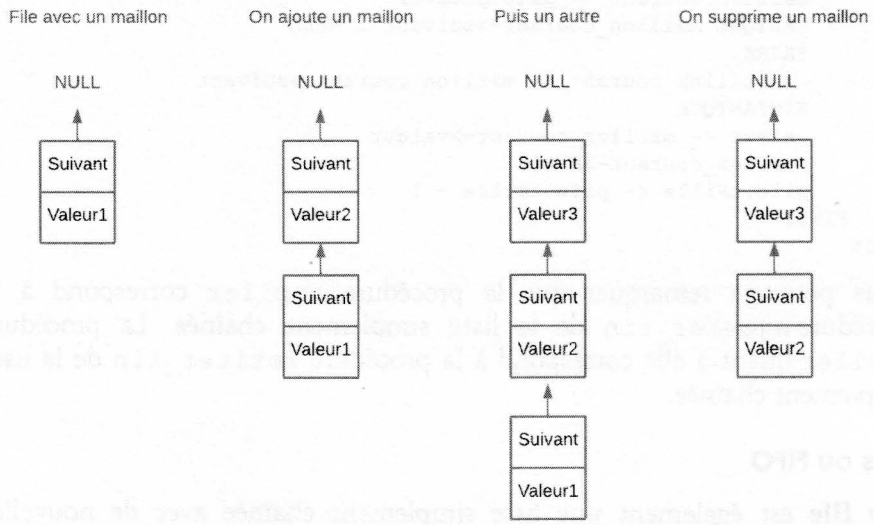
2.4.2 Files ou FIFO

Une **file** est également une liste simplement chaînée avec de nouvelles contraintes :

- L'ajout ne peut se faire qu'au début.
- La suppression ne peut se faire que sur le dernier.

Pour illustrer cette file, nous pouvons la comparer à l'attente d'une caisse au supermarché. Une fois toutes vos courses trouvées, vous devez payer en passant en caisse. Quand aucune caisse n'est libre, vous devez attendre votre tour : le premier client arrivé est le premier à passer donc à pouvoir partir après avoir payé.

Cela correspond à l'acronyme FIFO pour *First In Finst Out* : nous ne pouvons retirer que le premier élément ajouté, comme représenté sur la figure ci-dessous.



Fonctionnement d'une file

Les files sont utilisées par la machine pour mémoriser des tâches qui doivent être traitées selon leur ordre d'arrivée, modéliser des files d'attente ou encore gérer une mémoire tampon par exemple.

Comme nous l'avons vu avec la pile, nous pouvons utiliser à nouveau le `STRUCTURE` du maillon pour la file.

Les procédures d'ajout et de suppression sont remplacées par :

- La procédure `enfiler` qui prend en paramètres une file et une valeur, et qui ajoute cette valeur au début de la file.
- La procédure `defiler` qui prend en paramètres une file et une variable du même type que celui des données de la file et qui représentera la valeur retirée.

Implémentons la logique de ces procédures dans un algorithme :

```
STRUCTURE file_entier
DEBUT
   *premier <- NULL : maillon_entier
   taille <- 0 : ENTIER
FINSTRUCTURE

PROCEDURE enfiler(E/S : file : file_entier, E :  valeur : ENTIER)
VAR
   maillon <- NULL : maillon_entier
   maillon_courant <- NULL : maillon_entier
DEBUT
   maillon <- NOUVEAU maillon_entier
   maillon->valeur <- valeur
   SI file.taille > 0
   ALORS
      file.premier <- maillon
   SINON
      maillon_courant <- file.premier
      TANTQUE maillon_courant->suivant ≠ NULL
      FAIRE
         maillon_courant <- maillon_courant->suivant
      FINTANTQUE
      maillon_courant->suivant <- maillon
   FINSI
   file.taille <- file.taille + 1
FIN
PROCEDURE defiler(E/S: file: file_entier, E/S : valeur : ENTIER)
VAR
   *maillon : maillon_entier
DEBUT
   SI file.taille > 0
   ALORS
      valeur <- file.premier->valeur
      maillon <- file.premier
      file.premier <- file.premier->suivant
      maillon <- NULL
      file.taille <- file.taille 1 1
   FINSI
FIN
```

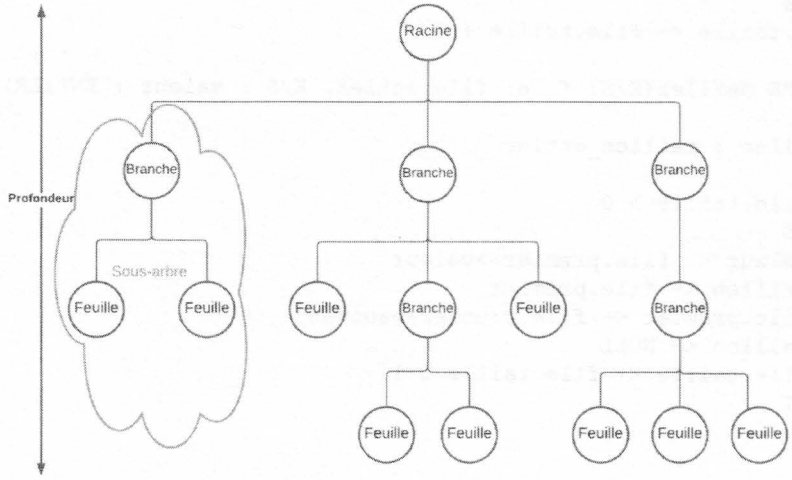
Nous pouvons remarquer que la procédure `enfiler` correspond à la procédure `ajouter_fin` de la liste simplement chaînée. La procédure `defiler` quant à elle correspond à la procédure `retirer_debut` de la liste simplement chaînée.

Nous avons fini l'étude des listes en algorithmie. Il nous reste maintenant un autre type de données complexe à voir : les arbres.

3. Les arbres

3.1 Principe

Un **arbre** en algorithmie est réellement inspiré des arbres de la nature. Ce sont des structures composées d'une racine, le point d'accès, ainsi que de branches et de feuilles, les points intermédiaires et terminaux. Nous ne pouvons parcourir un arbre qu'en partant de la racine. Chaque branche peut être vue comme un **sous-arbre**, comme illustré dans la figure ci-dessous, avec sa racine et ses branches. Cette définition nous inspire fortement une structure utilisant la récursivité.



Arbre quelconque

Nous utilisons, nous aussi, des arbres dans notre vie. L'exemple le plus flagrant pour illustrer cette structure complexe est l'arbre généalogique.

3.2 Création

L'arbre est créé grâce à une structure complexe de données qui, comme la liste, va se construire sur une autre `STRUCTURE` : les **nœuds**. Un nœud d'un arbre peut être la racine, une branche ou une feuille. Chaque nœud va donc posséder une liste d'autres nœuds, sauf les feuilles par lesquelles cette liste sera une liste vide.

Un arbre vide est donc une `STRUCTURE` contenant un pointeur sur un nœud initialisé à `NULL`.

Chaque arbre possède également une caractéristique décrivant son nombre de niveaux : la **profondeur**. Nous pouvons vulgariser la profondeur par le nombre de fois où nous pouvons descendre dans l'arbre. Par exemple, l'arbre de la figure précédente a une profondeur de trois. Un arbre ne contenant qu'une racine aura une profondeur de zéro.

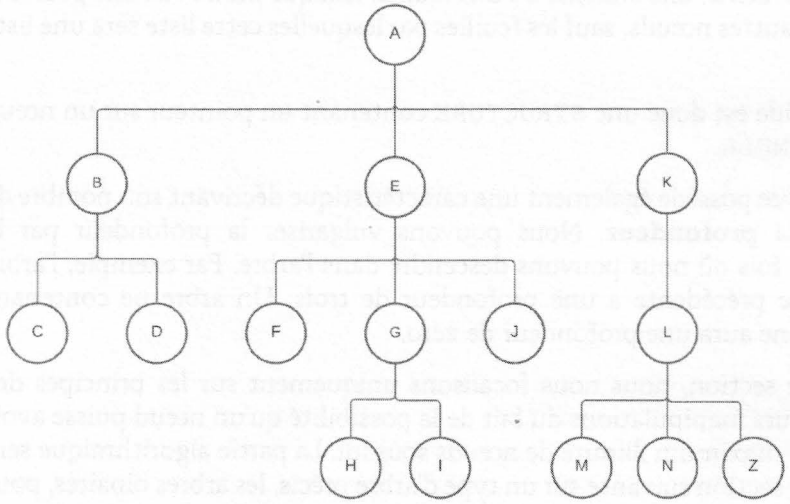
Dans cette section, nous nous focalisons uniquement sur les principes des arbres et leurs manipulations du fait de la possibilité qu'un nœud puisse avoir un nombre maximum illimité de nœuds sous lui. La partie algorithmique sera vue dans la section suivante sur un type d'arbre précis, les arbres binaires, pour des raisons pédagogiques.

Étudions maintenant les différents parcours des arbres.

3.3 Parcours en largeur

Le parcours en **largeur** d'un arbre consiste à parcourir l'arbre **par niveau**. Nous commençons par la racine puis par les branches de la racine, puis les branches des branches et ainsi de suite jusqu'aux feuilles.

Le parcours en largeur de l'arbre représenté sur la figure ci-dessous sera le suivant : A B E K C D F G J L H I M N et Z pour finir.



Exemple d'arbre pour illustrer les parcours

3.4 Parcours en profondeur

Le parcours en **profondeur**, appelé aussi **préfixe**, peut être vu comme l'opposé du parcours en largeur : nous parcourons après la racine sa première branche jusqu'à arriver à ses feuilles, puis la deuxième jusqu'à arriver à ses feuilles et ainsi de suite.

Le parcours en profondeur de l'arbre représenté sur la figure ci-dessus sera le suivant : A B C D E F G H I J K L M N puis Z pour finir.

3.5 Parcours en infixe

Pour le parcours en **infixe**, nous ne commençons pas par la racine mais par le sous-arbre le plus à gauche, puis la racine, puis le sous-arbre suivant et ainsi de suite.

Le parcours en infixe de l'arbre représenté sur la figure précédente sera le suivant : B C D A E F GH I J M N Z L et K pour finir.

3.6 Parcours en postfixe

Dans le parcours en **postfixe**, nous parcourons en premier les sous-arbres pour finir par la racine.

Le parcours en profondeur de l'arbre représenté sur la figure précédente sera le suivant : C D B H I G F J E M N Z L K et A pour finir.

3.7 Insertion d'une feuille

Pour ajouter une feuille, la logique est assez simple. Nous effectuons un parcours sur la branche considérée et nous ajoutons à sa feuille un nouveau nœud. L'ancienne feuille devient une branche, et le nouveau nœud la nouvelle feuille de cette branche.

3.8 Insertion d'une racine

Nous pouvons également ajouter un nouveau nœud à la racine. Le nouveau nœud deviendra la racine de l'arbre et l'ancienne racine, la branche de la nouvelle racine.

3.9 Insertion d'une branche

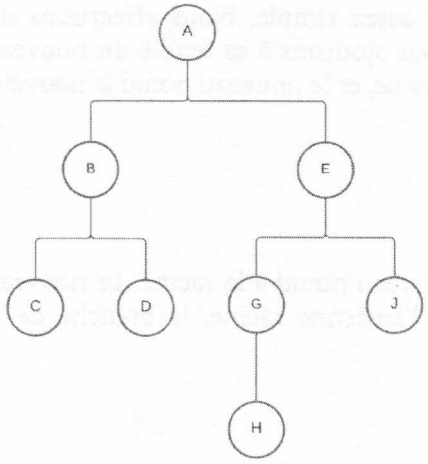
Pour insérer un nouveau nœud branche, donc non feuille et non racine, les choses se compliquent un peu. Imaginons que nous voulions ajouter un nœud Y entre le nœud L et Z de l'arbre. Pour ce faire, il nous faut parcourir la banche contenant L jusqu'à L, stocker temporairement le nœud Z, indiquer que le nouveau nœud sera le suivant de L et le précédent de Z.

3.10 Arbres binaires

3.10.1 Création

Travailler sur un arbre quelconque est très complexe en informatique du fait de la liste dynamique de nœuds pour représenter les branches de l'arbre.

Nous travaillons principalement sur des arbres binaires. Ils nous permettent de garder notre structure d'arbre mais en imposant une contrainte : une branche ne peut avoir au maximum que deux nœuds, comme le montre la figure ci-dessous.



Exemple d'arbre binaire

Grâce à cette contrainte, nous pouvons maintenant, de manière plus aisée, modéliser notre nœud et notre arbre en algorithmie. Nous choisissons de manipuler un arbre d'entiers dans la suite de cette section.

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
```

Avec ces deux structures, un arbre vide sera un arbre dont la racine vaut `NULL` et un arbre non vide sera un nœud ayant ou non un sous-arbre à gauche suivi ou non d'un sous-arbre à droite.

Remarque

Un arbre binaire équilibré est un arbre dont toutes les branches ont la même profondeur.

3.10.2 Parcours en largeur

Le parcours en largeur est un parcours par étage : celui de l'arbre représenté dans la figure de la section précédente donne le chemin A B E C D G J puis H.

Pour effectuer le parcours en largeur d'un arbre, nous aurons besoin d'une **file de nœuds** avec les opérations `enfiler` et `defiler` correspondantes. Nous devons donc créer une nouvelle structure pour les maillons de cette file. Pour ne pas écrire ces opérations, il nous suffit de typer le maillon avec le type du nœud.

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
```

```
   *racine <- NULL : noeud_entier
FINSTRUCTURE
STRUCTURE file_parcours_arbre
DEBUT
   premier <- NULL : maillon_parcours_arbre
   taille <- 0 : ENTIER
FIN
STRUCTURE maillon_parcours_arbre
DEBUT
   valeur <- NULL : noeud_entier
FINSTRUCTURE
PROCEDURE parcours_largeur(E : arbre : arbre_binaire_entier)
VAR
   parcours <- NULL : file_entier
   *noeud_temporaire <- NULL : noeud_entier
   valeur : ENTIER
DEBUT
   enfiler(parcours, arbre.racine)
   TANTQUE parcours.premier ≠ NULL
   FAIRE
      defiler(parcours, noeud_temporaire)
      ECRIRE(noeud_temporaire ->valeur)
      SI noeud_temporaire->noeud_gauche ≠ NULL
      ALORS
         enfiler(parcours, noeud_gauche)
      FINSI
      SI noeud_temporaire->noeud_droit ≠ NULL
      ALORS
         enfiler(parcours, noeud_droit)
      FINSI

   FIN
FIN
```

Nous parcourons notre arbre en commençant par la racine puis ses branches. L'astuce dans ce parcours est l'utilisation d'une file. Avec l'exemple de la figure précédente, nous enfilons A. Nous dépilons ensuite A pour l'afficher et enfilons B et E. Puis nous dépilons B pour l'afficher et enfilons C et D. Après nous dépilons E et enfilons G et J... ce qui nous donne bien un parcours en largeur.

3.10.3 Parcours en profondeur

Pour parcourir un arbre binaire en profondeur, nous utilisons le fait qu'une branche est également un arbre. Nous affichons en premier la racine de notre arbre et ensuite nous appelons la même procédure sur les sous-arbres de gauche puis sur le sous-arbre de droite.

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
PROCEDURE parcours_profondeur_arbre_binaire(E : arbre :
arbre_binaire_entier)
VAR
   arbre_temporaire : arbre_binaire_recherche
DEBUT
   SI arbre.racine ≠ NULL
   ALORS
      arbre_temporaire.racine <- arbre.racine
      ECRIRE(arbre_temporaire.racine->valeur)
      parcours_profondeur_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
      parcours_profondeur_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
   FINSI
FIN
```

Lorsque nous appliquons cette fonction sur l'arbre de la figure précédente, nous obtenons bien le chemin A B C D E F G H et J pour finir.

3.10.4 Parcours en infixe

Pour parcourir un arbre binaire en infixe, nous devons juste modifier l'ordre de la procédure du parcours en profondeur : d'abord la partie gauche de l'arbre puis sa racine puis la partie droite.

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
PROCEDURE parcours_infixe_arbre_binaire(E : arbre :
arbre_binaire_entier)
VAR
   arbre_temporaire : arbre_binaire_recherche
DEBUT
   SI arbre.racine ≠ NULL
   ALORS
      arbre_temporaire.racine <- arbre.racine
      parcours_infixe_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
      ECRIRE(arbre_temporaire.racine->valeur)
      parcours_infixe_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
   FINSI
FIN
```

3.10.5 Parcours en postfixe

Le parcours postfixe demande d'afficher en premier le sous-arbre de gauche puis celui de droite pour finir par la racine.

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
PROCEDURE parcours_postfixe_arbre_binaire(E : arbre :
arbre_binaire_entier)
VAR
   arbre_temporaire : arbre_binaire_recherche
```

```
DEBUT
   SI arbre.racine ≠ NULL
   ALORS
      arbre_temporaire.racine <- arbre.racine
      parcours_ postfixe_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
      parcours_ postfixe_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
      ECRIRE(arbre_temporaire.racine->valeur)
   FINSI
FIN
```

3.10.6 Ajout d'une feuille

Pour ajouter une feuille dans un arbre, il nous faut connaître son nœud parent pour savoir où l'ajouter. Il nous faudra donc un sous-programme pour trouver le parent. Pour le cas d'un ajout à un arbre vide, il nous suffit d'instancier la racine de l'arbre avec le nouveau nœud.

Nous partons des contraintes suivantes pour nos algorithmes :

- Le parent existe bien dans l'arbre.
- L'insertion se fait d'abord à gauche puis à droite.
- Si le parent a déjà deux branches, l'insertion échoue.

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
FONCTION chercher_noeud(arbre : arbre_binaire_entier,
valeur : ENTIER) : *noeud_entier
VAR
   arbre_temporaire : arbre_binaire_entier
DEBUT
   SI arbre.racine->valeur = valeur
   ALORS
      RETOURNER(arbre.racine)
   SINON
```

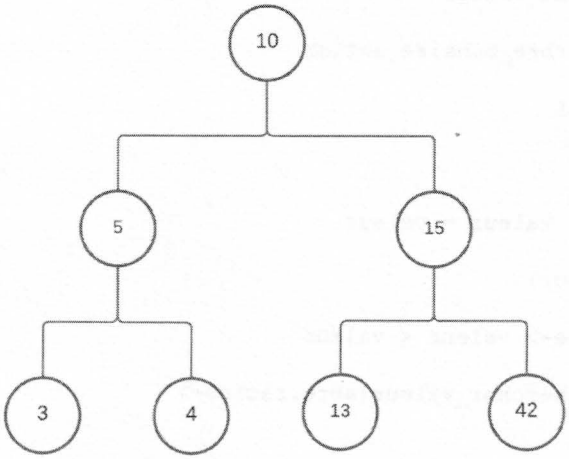
```
      SI arbre.racine->noeud_gauche ≠ NULL
      ALORS
         arbre_temporaire <- chercher_noeud(arbre, valeur)
         SI arbre_temporaire ≠ NULL ET
arbre_temporaire.racine->valeur == valeur
         ALORS
            RETOURNER(arbre_temporaire.racine)
         FINSI
      FINSI
      SI arbre.racine->noeud_droit ≠ NULL
      ALORS
         arbre_temporaire <- chercher_noeud(arbre, valeur)
         SI arbre_temporaire ≠ NULL ET
arbre_temporaire.racine->valeur == valeur
         ALORS
            RETOURNER(arbre_temporaire.racine)
         FINSI
      FINSI
   FINSI
FIN
PROCEDURE ajouter_noeud_arbre_binaire(E/S : arbre :
arbre_binaire_entier, E : valeur : ENTIER)
VAR
   *parent <- NULL : noeud_entier
   *ajout <- NULL : noeud_entier
DEBUT
   SI arbre.racine = NULL
   ALORS
      arbre.racine <- NOUVEAU noeud_entier
      arbre.racine->valeur <- valeur
   SINON
      parent <- CHERCHER(arbre, valeur)
      ajout <- NOUVEAU noeud_entier
      ajout->valeur <- valeur
      SI parent->noeud_gauche = NULL
      ALORS
         parent->noeud_gauche <- ajout
      SINON
         SI parent->noeud_droit = NULL
         ALORS
            parent->noeud_droit <- ajout
         FINSI
      FINSI
   FINSI
FIN
```

Pour rechercher une valeur, nous regardons si elle correspond à la valeur de l'arbre passé en paramètre. Si ce n'est pas le cas, nous allons la rechercher dans le sous-arbre gauche puis dans le sous-arbre à droite grâce aux appels récursifs de la fonction. Une fois le nœud parent trouvé, il suffit de créer la feuille au bon endroit (à gauche si la valeur est plus petite sinon à droite).

3.11 Arbres binaires de recherche

3.11.1 Principe

Un arbre binaire de recherche est un arbre binaire facilitant la recherche d'une valeur comme son nom le suggère. Pour simplifier cette opération, l'arbre binaire possède une contrainte sur les nœuds de l'arbre : les valeurs à gauche du nœud courant doivent être plus petites que la valeur du nœud et celles à droite plus grandes que la valeur du nœud, comme le montre la figure ci-après.



Exemple d'arbre binaire de recherche

Les structures `arbre_binaire_entier` et `noeud_entier` peuvent donc modéliser un arbre binaire de recherche, la seule différence sera pour l'ajout d'un nœud. Les procédures de parcours s'appliquent donc également aux arbres binaires de recherche.

3.11.2 Rechercher une valeur

Avec un arbre binaire, nous savons dès la racine où chercher notre valeur. Si elle est égale à la valeur de la racine, nous l'avons trouvé. Sinon, si elle est plus petite, nous la recherchons dans la partie de gauche et si elle est plus grande, nous parcourons le sous-arbre de droite. Si aucune racine traitée n'est égale à la valeur, cela veut dire que la valeur n'est pas dans l'arbre.

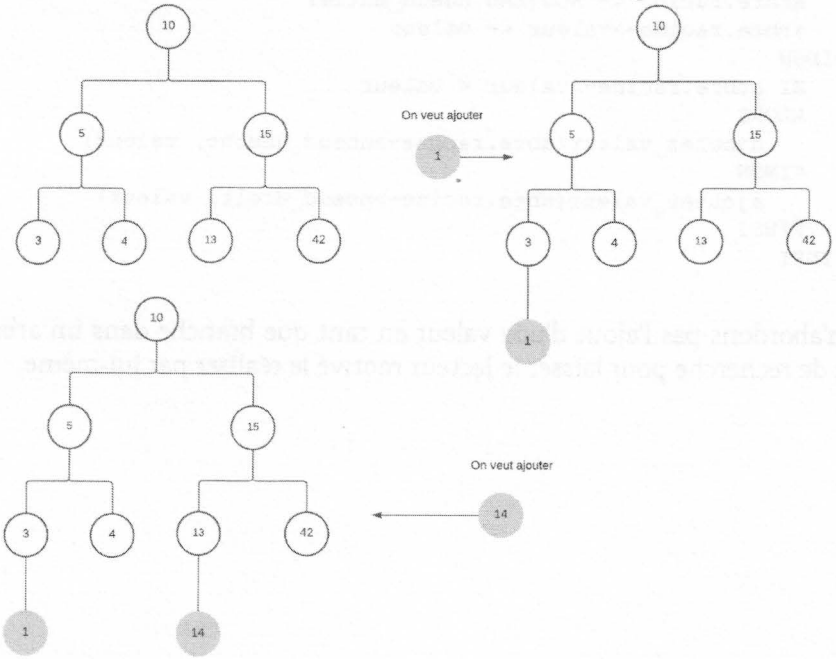
```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
    *racine <- NULL : noeud_entier
FINSTRUCTURE
FONCTION chercher_valeur(arbre : arbre_binaire_entier,
valeur : ENTIER) : *noeud_entier
VAR
   arbre_temporaire : arbre_binaire_entier
DEBUT
   SI abre.racine = NULL
   ALORS
      RETOURNER(NULL)
   SINON
      SI arbre.racine-> valeur = valeur
      ALORS
         RETOURNER(valeur)
      SINON
         SI arbre.racine-> valeur < valeur
         ALORS
            RETOURNER(chercher_valeur(abre.racine->
noeud_gauche, valeur))
         SINON
            RETOURNER(chercher_valeur(abre.racine->
noeud_droit, valeur))
         FINSI
      FINSI
   FINSI
FIN
```

La récursivité de cette fonction est une fois de plus élégante : si la racine n'est pas la valeur recherchée, nous appelons cette fonction sur le sous-arbre gauche si la valeur est plus petite que la valeur de la racine, sinon nous appelons cette fonction sur le sous-arbre de droite. Si nous avons parcouru tous les sous-arbres sans trouver la valeur recherchée, nous retournons `NULL` pour dire qu'elle n'existe pas dans l'arbre passé en paramètre.

3.11.3 Ajout d'une feuille

Lorsque l'élément à ajouter est une feuille de l'arbre, comme le montre la figure ci-dessous, le nœud parent, l'ancienne feuille donc, devient une branche avec comme feuille ce nouvel élément qui est placé à gauche ou à droite selon la valeur.

Pour un tel ajout, nous parcourons l'arbre à la recherche de la valeur à insérer en ajoutant le nouveau nœud à l'endroit où la recherche s'est arrêtée.



Exemples d'ajouts simples dans un arbre binaire de recherche

```
STRUCTURE noeud_entier
DEBUT
   *noeud_gauche <- NULL : noeud_entier
   *noeud_droit  <- NULL : noeud_entier
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
PROCEDURE ajouter_valeur(E/S  : arbre : arbre_binaire_entier,
E : valeur : ENTIER) : *noeud_entier
VAR
   arbre_temporaire : arbre_binaire_entier
DEBUT
   // nous sommes arrivés au bon endroit
   SI abre.racine = NULL
   ALORS
      arbre.racine <- NOUVEAU noeud_entier
      arbre.racine->valeur <- valeur
   SINON
      SI arbre.racine->valeur < valeur
      ALORS
         ajouter_valeur(abre.racine->noeud_gauche, valeur)
      SINON
         ajouter_valeur(abre.racine->noeud_droit, valeur)
      FINSI
   FINSI
FIN
```

Nous n'abordons pas l'ajout d'une valeur en tant que branche dans un arbre binaire de recherche pour laisser le lecteur motivé le réaliser par lui-même.

4. Et avec Python?

En ce qui concerne les listes, Python n'implémente pas les tableaux à taille fixe, il implémente directement ces structures complexes grâce au type de données `list` qui possède une allocation dynamique de la mémoire. La gestion des pointeurs est entièrement effectuée par le langage, ce qui simplifie largement les programmes.

Cependant, Python ne possède pas de type de données correspondant aux arbres par défaut.

Nous vous proposons d'implémenter ces types de données complexes dans le dernier chapitre de cet ouvrage grâce à la notion d'objet car Python ne possède pas le type `STRUCTURE`, ce langage utilise l'objet pour nous permettre de créer nos propres types de données complexes.

5. Exercices

5.1 Exercice 1

Avec les structures et procédures de ce chapitre, créez un sous-programme pour rechercher une valeur dans une liste simplement chaînée.

5.2 Exercice 2

Avec les structures et procédures de ce chapitre, créez un sous-programme pour rechercher une valeur dans une liste doublement chaînée.

5.3 Exercice 3

Avec les structures et procédures de ce chapitre, créez un sous-programme pour rechercher une valeur dans un arbre binaire.

Chapitre 8
Les fichiers

1. Le système de fichiers

1.1 Préambule

N'en avez-vous pas assez de devoir toujours entrer vos valeurs à la main ? Est-ce vraiment de cette manière que fonctionnent les programmes ?

Pour donner une solution à la première question, nous allons voir dans ce chapitre la **persistance** des données dans un fichier. La persistance indique juste que nous allons stocker physiquement sur l'ordinateur les données que le programme va utiliser mais aussi celles que le programme peut compiler.

Avant de voir comment stocker des informations dans des fichiers, il nous faut voir ce que sont justement ces fichiers.

1.2 Répertoire et fichier

Vous avez normalement l'habitude de gérer vos répertoires et fichiers avec un explorateur de fichiers de manière graphique. Sachez que cette gestion peut également se faire par un terminal ou une console, sinon aucun programme informatique ne pourrait accéder à des fichiers.

Un fichier est **un ensemble d'informations** réunies grâce à un même nom, enregistré sur un disque. Quel que soit le type de fichier, un fichier est un ensemble d'octets car le langage binaire est le seul langage que comprend la machine, comme la montre la figure ci-dessous représentant un fichier texte en octet. Cette figure représente le début de ce chapitre écrit avec le logiciel Word (Sublime Text permet d'accéder au fichier brut quel que soit le type de fichier).

```
 1  504b 0304 1400 0600 0800 0000 2100 ded4
 2  1d26 b901 0000 6309 0000 1300 0802 5b43
 3  6f6e 7465 6e74 5f54 7970 6573 5d2e 786d
 4  6c20 a204 0228 a000 0200 0000 0000 0000
 5  0000 0000 0000 0000 0000 0000 0000 0000
 6  0000 0000 0000 0000 0000 0000 0000 0000
 7  0000 0000 0000 0000 0000 0000 0000 0000
 8  0000 0000 0000 0000 0000 0000 0000 0000
 9  0000 0000 0000 0000 0000 0000 0000 0000
10  0000 0000 0000 0000 0000 0000 0000 0000
11  0000 0000 0000 0000 0000 0000 0000 0000
12  0000 0000 0000 0000 0000 0000 0000 0000
13  0000 0000 0000 0000 0000 0000 0000 0000
14  0000 0000 0000 0000 0000 0000 0000 0000
15  0000 0000 0000 0000 0000 0000 0000 0000
```

Fichier compris par l'ordinateur

Afin d'organiser les fichiers, les systèmes d'exploitation implémentent des fichiers spéciaux : les **répertoires ou dossiers**. Un répertoire est un fichier représentant une liste des fichiers qu'il contient. Ici, ces fichiers peuvent être des fichiers de données numériques ou d'autres répertoires.

Tout fichier possède au moins les propriétés suivantes :

- Un **nom unique** dans le répertoire pour l'identifier.
- Une **taille** en octets.
- Des **droits d'accès** pour le protéger.

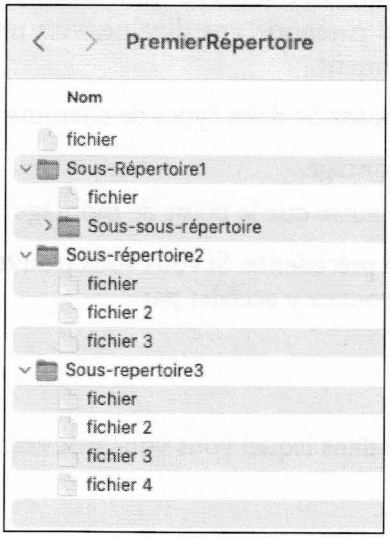
Les droits d'accès indiquent qui a le droit de faire quoi sur le fichier :

- Droit de **lecture** : pouvoir accéder aux données pour les lire.
- Droit **d'écriture** : pouvoir écrire dans le fichier.
- Droit **d'exécution** : pouvoir exécuter un fichier qui représente un programme.

La grande question est de savoir comment un programme peut retrouver un fichier sur un disque.

1.3 Arborescence de fichiers

Chaque fichier, répertoire ou non, est toujours dans un répertoire. Cette imbrication de fichiers représente donc une hiérarchie appelée **arborescence de fichiers**. En effet, comme le montre la figure ci-dessous, cette hiérarchie possède une racine, le premier répertoire, qui est suivie par des branches, les sous-répertoires et fichiers contenus.



Arborescence de fichiers

Cette racine est le **point de montage** du disque dur que vous parcourez :

- C: sur Window (système DOS)
- / sur Linux et macOS (système Unix)

C'est à partir de ce point de montage que vous pouvez récupérer un fichier. Prenons un exemple : vous travaillez sur le fichier exemple qui se trouve sur votre bureau. Pour y accéder, vous utilisez le chemin :

- /users/moi/Desktop/exemple sur Linux et macOS
- C:\Utilisateurs\moi\Bureau\exemple sur Windows

Nous pouvons déjà remarquer une différence notable entre les systèmes Unix et le système DOS : Unix utilise le / pour entrer dans un répertoire et DOS le \. Faites bien attention à ce point dans vos prochains programmes.

1.4 Chemin absolu ou relatif

Partir du point de montage peut s'avérer rébarbatif et délicat selon le nombre de répertoires à parcourir. Si vous devez en parcourir une dizaine, voir une vingtaine, vous risquez de vous perdre facilement.

Pour simplifier l'accès à un fichier, vous disposez de deux types de chemins :

- Le chemin **absolu** qui part du point de montage.
- Le chemin **relatif** qui part d'un répertoire autre que le point de montage.

Reprenons notre fichier *exemple* de la section précédente. Si vous vous trouvez déjà dans votre répertoire utilisateur, vous pouvez y accéder par :

- ./Desktop/exemple sur Linux et macOS
- ./Bureau\exemple sur Windows

Le "./" représente le répertoire courant, celui dans lequel vous vous trouvez.

Si vous êtes dans le répertoire sous-rep qui est sur le bureau, pour accéder à votre fichier, vous devez aller dans le répertoire qui contient le répertoire sous-rep. Pour remonter d'un répertoire, vous pouvez utiliser le raccourci ".." représentant le répertoire précédent :

- /../exemple sur Linux et macOS
- \..\exemple sur Windows

2. Les différents types de fichiers

Un fichier est défini par son nom ou son identifiant mais surtout par son **type**, représenté par son **extension**.

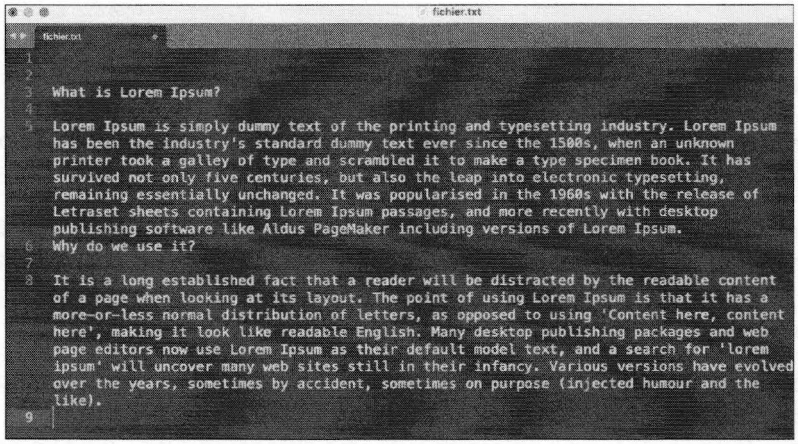
Le type du fichier permet d'indiquer à l'ordinateur comment sont organisées les informations, c'est-à-dire comment elles sont **formatées**. Avec cette indication, l'ordinateur peut également déterminer avec quelle application il doit ouvrir ou exécuter le fichier.

Prenons l'exemple de nos scripts Python. Leur extension est .py ce qui indique à l'interpréteur Python qu'il peut les interpréter. Si vous demandez à l'interpréteur Python de lancer un script avec une extension ".xls", il ne va pas comprendre le fichier et affichera une erreur.

Dans cette section, nous allons nous focaliser sur quelques exemples de types de fichiers test qui peuvent être manipulés par des programmes en commençant par les fichiers texte simples pour finir par quelques formats de fichiers texte utilisés en informatique.

2.1 Texte non formaté

Les fichiers texte non formatés ont généralement l'extension ".txt", ou aucune extension quelques fois. Ce sont les fichiers les plus simples : il s'agit uniquement de texte encodé principalement avec la table ASCII, comme le montre la figure ci-dessous :



Un fichier texte non formaté

Les encodages principaux des fichiers texte sont UTF-8 et le latin-1 (principalement sur macOS).

Vous pouvez ouvrir les fichiers textes non formatés avec des applications comme Bloc note ou Éditeur de texte.

2.2 Texte formaté

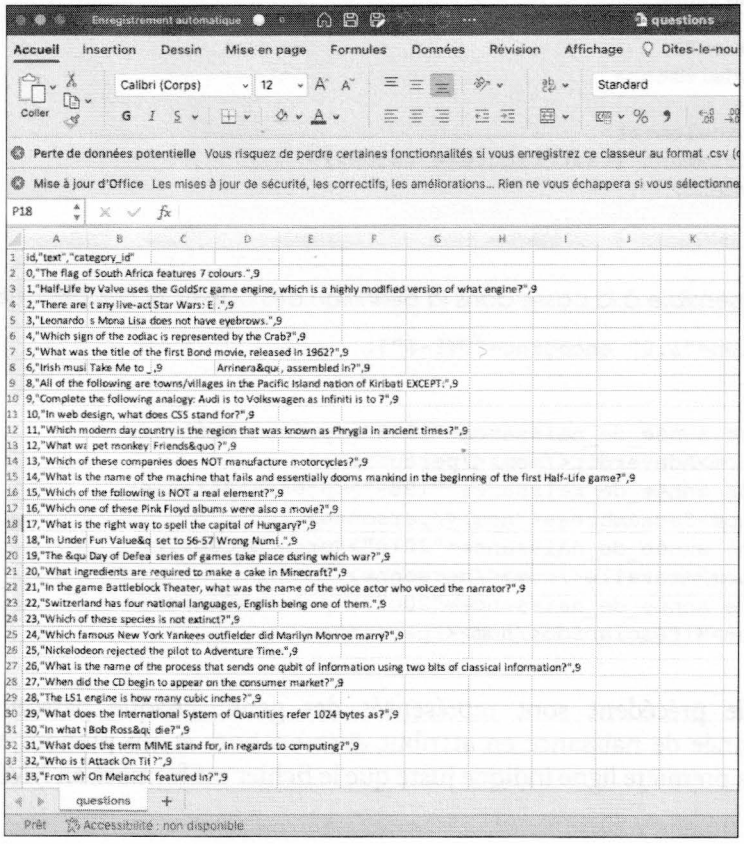
Le texte étant souvent utilisé pour communiquer avec des programmes en informatique, certains types de fichiers imposent un format pour les informations contenues. Cela permet au développeur de retrouver plus facilement les informations du fichier.

La plupart des manipulations de ces formats de fichiers texte formatés sont intégrées dans les langages de programmation aujourd'hui.

2.2.1 CSV

Le format **CSV**, ou *Coma-Separated Values*, permet de séparer les champs d'une STRUCTURE par des virgules et les STRUCTURES par des sauts de ligne.

Comme le montre la figure ci-dessous, les fichiers .csv sont manipulables avec un tableur pour un être humain.



Un exemple de fichier CSV

2.2.2 XML

Les fichiers **XML**, pour e*X*t*ensible Markup Language*, donnent une structure sous forme de **balises** aux données.

Un fichier XML est toujours une balise contenant d'autres balises. Une balise est définie par un identifiant unique au sein du fichier, entre chevrons. Les valeurs des données peuvent être valorisées soit en **attribut** de la balise, soit à **l'intérieur** de la balise.

```
<myfirstelement>
   <mysecondelement>
      value
   </mysecondelement>
   <mythirdelement itsattribute=« value»/>
</myfistelement>
```

Remarque

Le format est sensible à la casse dans la définition des balises et des attributs.

```
<?xml version="1.0" encoding="UTF-8"?>

<gens>
   <personne année_de_naissance="1988"><prénom>Adélaïde
</prénom><nom>Adeverdit</nom></personne>
   <personne année_de_naissance="1989"><prénom>Béranger
</prénom><nom>Sachambress</nom></personne>
   <personne année_de_naissance="2010"><prénom>Cédric
</prénom><nom>Azaraile</nom></personne>
   <personne année_de_naissance="2007"><prénom>Désirée<
/prénom><nom>Mifasolacido</nom></personne>
</gens>
```

Dans l'exemple précédent sont représentés des gens : chaque personne possède une année de naissance en attribut avec un nom et un prénom en sous-balises. La première ligne indique juste que le fichier est formaté en XML.

2.2.3 JSON

Le XML étant très verbeux avec les identifiants des balises, donc lourd pour envoyer des informations d'un programme à un autre, un format de fichier texte le remplace de nos jours : le JSON pour *JavaScript Object Notation*.

Remarque

Même si le JSON a pour objectif de remplacer le XML, le format XML reste un standard dans l'informatique et ne pourra pas être obsolète avant de nombres années.

Le principe du JSON est de garder les informations d'un fichier XML en enlevant au maximum les identifiants. Ainsi une liste sera définie par une paire de crochets, ses éléments étant séparés par des virgules. Cette liste remplace la première balise du fichier XML et chaque balise contenant des sous-balises. Chaque autre type d'information est donné avec le format clé : valeur, comme dans un dictionnaire. Les données de type chaîne de caractères sont également entre double quotes comme en algorithmie.

Transformons notre fichier XML en JSON :

```
{
   personne : {
      annee_de_naissance : 1988 ,
      prenom : "Adélaïde " ,
      nom : "Adeverdit ",
   },
   personne : {
      annee_de_naissance : 1989,
      prenom : "Béranger " ,
      nom : "Sachambress",
   },
   personne : {
      annee_de_naissance : 2010,
      prenom : "Cédric" ,
      nom : "Azaraile ",
   },
   personne : {
      annee_de_naissance : 2007,
      prenom : " Désirée " ,
      nom : " Mifasolacido ",
   }
}
```

Le format JSON paraît plus complexe à appréhender que le XML à première vue mais rassurez-vous, après quelques manipulations, vous serez à l'aise avec ce format qui deviendra naturel.

Remarque

Pour aller plus loin avec la persistance des données dans un programme en informatique, nous pouvons conseiller au lecteur de s'intéresser aux bases de données et au langage SQL avec un ouvrage tel que SQL, Les fondamentaux du langage écrit par Anne-Christine Bisson et publié aux Éditions ENI. Les bases de données sont une technologie qui permet de formater simplement les données tout en les enregistrant sur le disque et qui, pour finir, est facilement intégrable dans vos programmes.

3. Manipulation de fichiers

Pour des raisons de simplicité, nous ne manipulerons que des fichiers texte non formatés dans le reste de ce chapitre. Nous laissons le lecteur étudier la manipulation des autres types de formats de fichiers par lui-même. En effet, nous lui apportons les bases nécessaires à la manipulation de tout type de fichier texte.

De plus, nous partons également du principe que le fichier manipulé existe sur le disque et se trouve dans le même répertoire que le fichier de l'algorithme.

3.1 Ouvrir et fermer un fichier

Manipuler un fichier revient à utiliser un pointeur, appelé **curseur** pour les fichiers. Qui dit pointeur, dit gestion de son allocation et de sa désallocation.

Ce pointeur est une variable de type `FICHIER`. Pour l'allouer, nous allons ouvrir le fichier avec la fonction `OUVRIR-FICHIER` et pour le supprimer, nous allons fermer le fichier avec la fonction `FERMER-FICHIER`.

La fonction OUVRIR-FICHIER prend deux paramètres :

- Le **nom** du fichier à ouvrir.
- Le **mode** d'ouverture :
  - ecriture pour pouvoir écrire dans le fichier en commençant par la première ligne.
  - lecture pour pouvoir récupérer les données du fichier sous forme de chaînes de caractères.

```
VAR
   mon_fichier : FICHIER

mon_fichier <- OUVRIR-FICHIER("nom du fichier", "mode d'ouverture")
```

La fonction FERMER prend comme paramètre le fichier de type FICHIER à fermer.

```
FERMER-FICHIER(mon_fichier)
```

```
PROGRAMME Ouverture_fermeture_fichier
VAR
   mon_fichier : FICHIER
   nom_du_fichier : CHAINE
DEBUT
   ECRIRE("Entrez le nom du fichier")
   nom_du_fichier <- LIRE()
   mon_fichier <- OUVRIR-FICHIER(nom du fichier, "lecture")
   FERMER-FICHIER(mon_fichier)
FIN
```

Si vous ne fermez pas le fichier en fin d'algorithme ou de programme, vous laissez le curseur en place dans le fichier. Cela peut provoquer de graves erreurs : vous pouvez empêcher les modifications du fichier par un autre programme ou utilisateur, votre fichier ne sera donc plus à jour. Un fichier non fermé est également un fichier que le système d'exploitation ne peut plus manipuler : un fichier ouvert est un fichier qu'on ne peut ni supprimer ni déplacer.

3.2 Lire un fichier

Une fois le fichier ouvert, vous pouvez commencer à le lire. Sa lecture se fait de manière **séquentielle** : l'algorithme lit les lignes les unes après les autres.

Pour lire une ligne d'un fichier, nous allons utiliser la procédure `LIRE` qui prend en paramètre d'entrée le fichier de type `FICHIER` et en sortie la ligne lue de type `CHAINE`.

```
LIRE(mon_fichier, ligne)
```

Pour pouvoir parcourir un fichier, il faut savoir quand le fichier est fini. C'est le rôle de la fonction `EOF(mon-fichier)` (pour *End Of File*) : cette fonction retourne un booléen `VRAI` si nous sommes à la fin du fichier (le curseur sur le fichier ne pointe plus de texte) ou `FAUX` quand il reste des lignes dans le fichier.

Mettons en place l'algorithme pour lire un fichier choisi par l'utilisateur :

```
PROGRAMME Lecture_fichier
VAR
   mon_fichier : FICHIER
   nom_du_fichier : CHAINE
   ligne : CHAINE
DEBUT
   ECRIRE("Entrez le nom du fichier")
   nom_du_fichier <- LIRE()
   mon_fichier <- OUVRIR-FICHIER(nom du fichier, "lecture")
   TANTQUE NON EOF(mon_fichier)
   FAIRE
      LIRE(mon_fichier, ligne)
      ECRIRE(ligne)
   FINTANTQUE
   FERMER-FICHIER(mon_fichier)
FIN
```

La fonction `EOF` retournant un booléen, il est tout à fait logique d'utiliser une structure itérative `TANTQUE` car nous testons la validité d'une condition pour lire un fichier. Nous ne pouvons pas utiliser la structure `REPETER JUSQU'A` car nous ne savons pas au départ si le fichier contient du texte ou non.

3.3 Écrire dans un fichier

Vous pouvez également stocker des informations dans un fichier en écrivant à l'intérieur avec la fonction ECRIRE.

```
ECRIRE(fichier, ligne)
```

La fonction ECRIRE prend en paramètre le fichier de type FICHIER dans lequel écrire et la ligne de type CHAINE à écrire dans le fichier. Pour chaque ligne écrite, cette fonction ajoute à la fin un saut de ligne, exactement comme pour l'affichage console.

Mettons en place l'algorithme qui écrit dix lignes dans un fichier choisi par l'utilisateur puis qui les lit :

```
PROGRAMME Ecriturle_ecture_fichier
VAR
   mon_fichier : FICHIER
   nom_du_fichier : CHAINE
   ligne : CHAINE
   i : ENTIER
DEBUT
   ECRIRE("Entrez le nom du fichier")
   nom_du_fichier <- LIRE()
   mon_fichier <- OUVRIR-FICHIER(nom_du_fichier, "ecriture")
   POUR i ALLANT DE 1 A 10 AU PAS DE 1
   FAIRE
      ECRIRE("Entrez la ligne, ", i)
      ligne <- LIRE()
      ECRIRE(mon_fichier, ligne)
   FINPOUR
   FERMER-FICHIER(mon_fichier)

   mon_fichier <- OUVRIR-FICHIER(nom_du_fichier, "lecture")
   TANTQUE NON EOF(mon_fichier)
   FAIRE
      LIRE(mon_fichier, ligne)
      ECRIRE(ligne)
   FINTANTQUE
   FERMER-FICHIER(mon_fichier)
FIN
```

Nous remarquons que pour manipuler un fichier en écriture et en lecture dans le même algorithme, nous devons l'ouvrir et le fermer deux fois, une fois par mode d'accès.

3.4 Pour aller plus loin

Manipuler un fichier au sens de sa lecture ou de son écriture n'est pas si difficile car ces opérations sont déjà implémentées. La difficulté dans la gestion des fichiers et l'algorithmie vient du parcours de l'arborescence pour trouver le bon fichier à partir d'un répertoire donné.

Pour cela, nous devons commencer par parcourir tous les fichiers du répertoire, puis tous les sous-répertoires du répertoire, puis les sous-répertoires des sous-répertoires, etc. Notre intuition nous dit bien ici que nous allons devoir faire un algorithme récursif pour parcourir l'ensemble d'un répertoire donné.

Implémentons cet algorithme pour afficher le contenu d'un répertoire et de ses sous-répertoires. Utilisons le type de donnée `REPETOIRE` pour représenter les répertoires avec l'opération `LISTER` qui retourne un tableau contenant chaque fichier du répertoire. Il nous est également fourni l'opération `ETRE-FICHIER` qui nous indique si le fichier est un simple fichier ou un répertoire.

```
FONCTION parcourir-repertoire( mon_repertoire : REPERTOIRE)
VAR
   i, taille : ENTIER
   fichiers : TAB[1 … taille]: FICHIER
DEBUT
   LISTER(mon_repertoire, fichiesr, i)
   POUR i ALLANT DE 1 A taille AU PAS DE 1
   FAIRE
      SI ETRE-FICHIER(fichiers[i])
      ALORS
         ECRIRE("Fichier : ", fichiers[i])
      SINON
         ECRIRE("Sous-répertoire : ", fichiers[i])
         parcourir-repertoire(fichier[i])
      FINSI
   FINPOUR
FIN
```

4. Accédons à des fichiers avec Python

4.1 Ouvrir et fermer un fichier

Comme en algorithmie, si vous voulez manipuler un fichier, vous devez l'ouvrir. En Python, cette opération se fait avec la fonction `open`.

```
mon_fichier = open("nom du fichier", "mode", "encodage")
```

L'encodage est optionnel et représente l'encodage des caractères du fichier, par exemple UTF8.

Le **mode** représente les actions possibles sur le fichier :

- Accès en lecture : **r**.
- Accès en écriture : **w**.
- Accès en ajout : **a**.

La différence entre l'accès en écriture et celui en ajout est assez implicite. En écriture, vous écrivez dès le début du fichier, quitte à effacer ce qui est déjà présent dans le fichier. En ajout, vous vous placez d'office en fin de fichier pour ajouter des informations à la fin de ce dernier sans rien effacer du contenu.

Remarque

À la différence de l'algorithmie, si le fichier n'existe pas, Python va le créer lors de l'instruction open, en écriture ou en ajout. L'ouverture d'un fichier inexistant en mode Lecture provoque une erreur terminant le script.

Toute manipulation de fichier en Python impose de le fermer après utilisation, afin de ne pas provoquer d'incohérence ou de blocage sur ce fichier. Pour ce faire, en Python, il faut appeler la fonction `close` sur la variable représentant le fichier.

```
mon_fichier.close()
```

4.2 Lire un fichier

En Python, la lecture d'un fichier peut se faire grâce à trois fonctions qui retournent toutes une chaîne :

- Lire tout le fichier d'un seul coup : `read()`.
- Lire une ligne du fichier : `readline()`.
- Lire toutes les lignes du fichier (retourne une liste de chaînes) : `readlines()`.

Voyons maintenant un exemple d'utilisation de chacune de ces fonctions.

4.2.1 read()

Avec la fonction `read`, vous lisez tout le texte du fichier, saut de ligne inclus, en une instruction.

```
name =  input("Entrer le nom de votre fichier ")
fichier = open(name,'r', encoding="UTF-8")
texte = fichier.read()
print(texte)
fichier.close()
```

4.2.2 readline()

La fonction `readline` vous permet de lire votre fichier ligne par ligne.

```
name =  input("Entrer le nom de votre fichier ")
fichier = open(name,'r', encoding="UTF-8")
premiere_ligne = fichier.readline()
fichier.close()
```

Notons que parcourir de manière sûre notre fichier est délicat avec cette fonction.

```
Entrer le nom de votre fichier chapitre8.py
from math import fsum, pi
```

4.2.3 readlines()

La fonction `readlines` lit toutes les lignes du fichier et les stocke dans une liste de chaîne de caractères.

```
name =  input("Entrer le nom de votre fichier ")
fichier = open(name,'r', encoding="UTF-8")
lignes = fichier.readlines()

for ligne in lignes :
   print(ligne)

fichier.close()
```

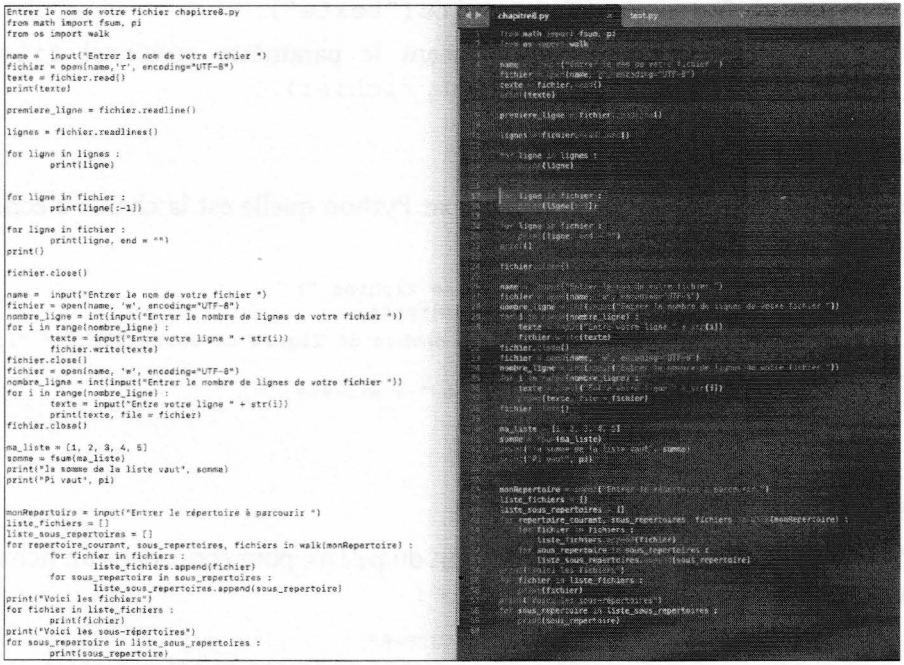
```
Entrer le nom de votre fichier chapitre8.py
from math import fsum, pi

from os import walk


name =  input("Entrer le nom de votre fichier ")

fichier = open(name,'r', encoding="UTF-8")

texte = fichier.read()

print(texte)


premiere_ligne = fichier.readline()


lignes = fichier.readlines()


for ligne in lignes :

        print(ligne)




for ligne in fichier :

        print(ligne[:-1])


for ligne in fichier :

        print(ligne, end = "")

print()


fichier.close()
```

4.2.4 for in

Vous pouvez également utiliser l'instruction `for in` qui parcourt chaque ligne du fichier. Cette méthode reste la plus simple en Python.

```
name =  input("Entrer le nom de votre fichier ")
fichier = open(name,'r', encoding="UTF-8")
```

```
for ligne in fichier :
    print(ligne[:-1])

fichier.close()
```



Notons que, lors de notre parcours de ligne, la ligne lue dans le fichier se termine par un saut de ligne : nous n'affichons donc pas son dernier caractère.

Une alternative est d'enlever le saut de ligne de la fonction `print` :

```
name =  input("Entrer le nom de votre fichier ")
fichier = open(name,'r', encoding="UTF-8")

for ligne in fichier :
    print(ligne, end = "")
print()

fichier.close()
```

4.3 Écrire dans un fichier

En Python, l'écriture dans un fichier peut se faire grâce à deux fonctions :

- Avec une chaîne en paramètre : `write("texte")`.
- Avec la fonction `print` en ajoutant le paramètre optionnel `file` : `print("mon texte", file=mon_fichier)`.

4.3.1 write(chaîne)

Le `write` indique au fichier ouvert par Python quelle est la chaîne à écrire dans ce fichier.

```
name =  input("Entrer le nom de votre fichier ")
fichier = open(name, 'w', encoding="UTF-8")
nombre_ligne = int(input("Entrer le nombre de lignes de votre fichier "))
for i in range(nombre_ligne) :
   texte = input("Entrer votre ligne " + str(i))
   fichier.write(texte)
fichier.close()
```

4.3.2 print()

Vous pouvez changer la sortie par défaut du `print` pour écrire dans un fichier et non sur la console avec l'option `file` :

```
fichier = open(name, 'w', encoding="UTF-8")
nombre_ligne = int(input("Entrer le nombre de lignes de votre fichier "))
for i in range(nombre_ligne) :
   texte = input("Entrer votre ligne " + str(i))
   print(texte, file = fichier)
fichier.close()
```

Remarque

Toutes nos écritures de fichiers se positionnent sur la première du ligne au risque d'effacer ce qui existe déjà dans le fichier. Si vous voulez converser le texte du fichier, vous devez utiliser le mode a pour append.

4.4 Parcourir l'arborescence

4.4.1 Module

Un module en Python permet d'importer et d'utiliser des fonctions définies dans ce module. Pour utiliser ses fonctions, vous devez importer le module.

```
import nomModule
```

```
from nomModule import laFonction, laDeuxiemeFonction
```

Le premier `import` importe toutes les fonctions définies dans le module. Pour les utiliser, il faudra les préfixer du nom du module suivi d'un point. Le second n'importe que les fonctions listées dans l'instruction. Vous pouvez les utiliser sans les préfixer.

Les imports sont toujours les premières instructions de votre script.

Illustrons les modules avec le module `math` qui contient la fonction `fsum` calculant la somme d'une liste et la constante `pi`.

```
import math
ma_liste = [1, 2, 3, 4, 5]
somme = math.fsum(ma_liste)
print("la somme de la liste vaut", somme)
print("Pi vaut", math.pi)
```

```
from math import fsum, pi
ma_liste = [1, 2, 3, 4, 5]
somme = fsum(ma_liste)
print("la somme de la liste vaut", somme)
print("Pi vaut", pi)
```

Remarque

Un module en Python peut contenir énormément d'instructions. Lors de son import, l'interpréteur Python lit tout le module. Il est donc conseillé de préférer le deuxième import pour ne pas surcharger la mémoire.

Remarque

Cette méthode peut également provoquer un conflit de nommage avec vos propres instructions. Par exemple, vous avez défini une fonction `sum` et vous utilisez également la fonction `sum` de Math. Deux solutions s'offrent à vous : renommer votre fonction ou préfixer la fonction du module par le nom du module.

4.4.2 Utilisation de walk

Pour manipuler l'arborescence de fichiers, Python propose le module `os` qui implémente toutes les fonctions nécessaires pour gérer les fichiers et les répertoires d'un disque.

Parmi ces fonctions, il existe la fonction `walk`. En l'appliquant sur un répertoire, elle va récupérer tous les répertoires, les sous-répertoires et tous les fichiers contenus dans le répertoire passé en paramètre.

```
from os import walk
monRepertoire = input("Entrer le répertoire à parcourir ")
liste_fichiers = []
liste_sous_repertoires = []
for repertoire_courant, sous_repertoires,
fichiers in walk(monRepertoire) :
   for fichier in fichiers :
      liste_fichiers.append(fichier)
   for sous_repertoire in sous_repertoires :
      liste_sous_repertoires.append(sous_repertoire)
print("Voici les fichiers")
for fichier in liste_fichiers :
   print(fichier)
print("Voici les sous-répertoires")
for sous_repertoire in liste_sous_repertoires :
   print(sous_repertoire)
```

5. Exercices

5.1 Exercice 1

Écrivez l'algorithme qui génère automatiquement dans un fichier texte les tables de multiplication bien écrites de 1 à 20. Codez le script Python correspondant.

5.2 Exercice 2

Écrivez l'algorithme qui recopie un fichier texte dont le nom est donné par l'utilisateur dans un autre ficher texte nommé "copie.txt". Codez le script Python correspondant.

5.3 Exercice 3

Écrivez l'algorithme qui compare deux fichiers texte et qui affiche la première différence. Codez le script Python correspondant.

5.4 Exercice 4

Écrivez l'algorithme qui recherche et affiche la ligne la plus longue dans un fichier texte. Codez le script Python correspondant.

5.5 Exercice 5

Codez le script Python qui remplace tous les espaces d'un fichier texte par un triple espace. Par exemple, "le chat" deviendra "le chat".

5.6 Exercice 6

Codez le script Python qui compte le nombre d'occurrences de chaque caractère d'un fichier texte en utilisant un dictionnaire.

5.7 Pour aller plus loin

5.7.1 Exercice 7

Codez le script Python qui liste tous les fichiers d'un répertoire dont le nom est donné par l'utilisateur.

5.7.2 Exercice 8

Codez le script Python qui demande à l'utilisateur de donner le nom d'un répertoire à parcourir afin de savoir si ce répertoire contient un fichier ou un répertoire dont le nom est également entré par l'utilisateur.

Chapitre 9
Commencer avec l'objet

1. Préambule

L'objectif de ce livre est d'apprendre la logique de programmation afin de pouvoir programmer vos premiers scripts en Python. Tout apprentissage du développement informatique commence par ce que nous avons vu jusqu'à présent : la programmation impérative, ou procédurale si vous préférez.

Ce style de programmation est à la base de toutes les autres, que ce soit la programmation fonctionnelle ou la **Programmation Orientée Objet** (**POO** ou *Object-Oriented Programming* en anglais), les deux grands courants actuels du développement informatique.

À l'heure de l'écriture de ce livre, nous pouvons voir un style de programmation revenir sur le devant de la scène : la programmation fonctionnelle. Il s'agit d'un style utilisé quasiment à la naissance de la machine et commençant à être implémenté dans la plupart des langages. La programmation fonctionnelle moderne en est encore à ses débuts et elle est le plus souvent utilisée pour traiter des masses de données comme en Big Data par exemple, c'est pourquoi nous ne l'introduisons pas dans cet ouvrage.

La programmation orientée objet détient facilement aujourd'hui plus de 80 % des parts du marché du développement informatique. Elle est apparue au début des années 1990, apportant une vraie révolution sur la modélisation des données dans les programmes.

Afin que vous puissiez continuer votre apprentissage avec de solides bases, nous vous proposons dans ce chapitre une introduction à la programmation orientée objet et ses principes de base.

Le lecteur intéressé par ce type de programmation pourra approfondir ses connaissances et compétences avec des ouvrages tels que UML 2.5 et Design Patterns ou encore Apprendre la Programmation Orientée Objet avec le langage Python dans la collection Ressources informatiques aux Éditions ENI.

2. Le naturel de l'objet

2.1 Introduction

La programmation orientée objet découle de la pensée humaine. C'est une technique de programmation de haut niveau qui permet aux développeurs de modéliser leurs données comme dans la vie réelle.

Par cela, nous voulons dire que ce type de programmation est inspiré des objets de la vraie vie. Prenons un simple objet comme une bouteille d'eau. Chaque bouteille d'eau a ses propres particularités tout en restant au fond une bouteille d'eau. En programmation orientée objet, nous pouvons traduire cet exemple avec une classe `BouteilleEau` qui décrit les **propriétés** de chaque bouteille d'eau : sa contenance et si elle est minérale et/ou gazeuse. Ainsi, avec cette classe, nous pouvons créer plusieurs bouteilles d'eau, nos **objets**, qui auront tous une valeur donnée pour leur contenance et leur type d'eau.

Plus simplement, voyez les classes comme l'évolution des STRUCTURES et les objets comme l'évolution des ENREGISTREMENTS. Les structures, comme les classes, permettent de lier différentes variables dans un même nouveau type de données. Les classes lient en plus les traitements et les variables sur lesquels ils portent. Une structure est instanciée avec un enregistrement et une classe avec un objet.

La programmation orientée objet va beaucoup loin qu'une simple évolution des `STRUCTURES`, elle permet d'avoir un **code mieux organisé**, c'est-à-dire organisé de manière plus intuitive pour l'humain, tout **en factorisant un maximum de code**.

2.2 L'objet et la classe

La classe nous permet de modéliser nos données dans un algorithme ou dans un programme avec une nouvelle organisation.

Nous regroupons les champs d'une structure en tant qu' **attributs** de la classe et les fonctions et procédures qui manipulent ces champs comme **méthodes** de la classe.

Vous pouvez voir la classe comme un nouveau type de donnée complexe que vous créez selon vos besoins.

À partir de la classe, nous pouvons créer un objet. Il s'agit de la variable du type de la classe. Nous n'utilisons pas le mot variable pour l'objet pour le différencier des variables simples dans nos algorithmes.

Reprenons notre exemple de bouteille d'eau. Une classe se définit par le mot-clé `CLASSE` et les variables liées à la classe par un bloc `ATTRIBUT`.

```
PROGRAMME Decouverte_classe_objet
CLASSE Bouteille
ATTRIBUT
DEBUT
   contenance : ENTIER
   minerale : BOOLEEN
   gazeuse : BOOLEEN
FIN
DEBUT
VAR
   monObjet : Bouteille
DEBUT
   // instanciation de notre objet de type Bouteille
   // Création de l'objet en initialisant ses attributs
   ECRIRE("Entrez la contenance de votre bouteille d'eau?")
   monObjet.contenance <- LIRE()
   ECRIRE("Est-elle minérale?")
   monObjet.minerale <- LIRE()
   ECRIRE("Est-elle gazeuse?")
   monObjet.gazeuse <- LIRE()
FIN
```

L'accès aux attributs d'un objet se fait comme pour les structures avec l'opérateur "." pour notre exemple qui est incomplet.

Remarque

*Pour la programmation orientée objet, nous utilisons la convention de nommage **CamelCase** : les classes commencent toujours par une majuscule et la séparation des mots d'un identifiants est également représenté par une majuscule.*

Remarque

*Attention à ne pas confondre classe et objet : **classe = type** et **objet = variable**.*

Transformons maintenant notre structure pour notre liste de courses.

```
STRUCTURE produit
DEBU
   nom : CHAINE
   quantité : ENTIER
FINSTRUCTURE
```

Devient :

```
CLASSE Produit
ATTRIBUT
DEBUT
   quantite : ENTIER
   nom : CHAINE
FIN
```

Nous remarquons qu'il n'y a pas de différence notable entre une `STRUCTURE` et une classe sur la modélisation des attributs. Cependant, les classes ajoutent les traitements sur ses attributs dans son bloc de méthodes.

2.3 Méthodes

Une classe permet de définir des fonctions et procédures qui lui appartiennent : les **méthodes**.

Les méthodes ont l'avantage de connaître les attributs de la classe, de pouvoir les manipuler sans les passer en paramètres. Étant intégrées à la classe, c'est à l'objet de les appeler par l'opérateur "."

Continuons notre liste de courses en programmation orientée objet, en ajoutant des méthodes à nos produits.

```
CLASSE Produit
ATTRIBUT
DEBUT
   quantite : ENTIER
   nom : CHAINE
FIN
METHODE
DEBUT
   PROCEDURE diminuer_quantite(E : combien : ENTIER)
   DEBUT
      quantite <- quantite - combien
   FIN

   PROCEDURE produit_achete()
   DEBUT
      quantite <- 0
   FIN
FIN
```

Maintenant, pour diminuer la quantité d'un produit, au lieu de changer son champ, nous allons appeler la méthode `diminuer_quantite` :

```
mon_produit.diminuer_quantite(2)
```

2.4 Visibilité des attributs et méthodes

Afin de protéger l'appel de certaines méthodes ou la modification de certains attributs, la programmation orientée objet nous permet d'en définir la **visibilité** :

- `PUBLIC` : l'attribut ou la méthode est accessible dans tout le programme.
- `PRIVE` : l'attribut ou la méthode n'est accessible que par l'objet instanciant la classe (en vulgarisant, accessible que dans la classe).

De ce fait, chaque classe va pouvoir déclarer deux blocs pour les attributs, un pour les privés et un pour les publiques, et de même pour les méthodes, comme le montre l'évolution de notre classe `Produit`.

```
CLASSE Produit
ATTRIBUT PUBLIC
DEBUT
   quantite : ENTIER
FIN
ATTRIBUT PRIVE
DEBUT
   nom : CHAINE
FIN
METHODE PUBLIC
DEBUT
   PROCEDURE diminuer_quantite(E : combien : ENTIER)
   DEBUT
      quantite <- quantite - combien
   FIN
FIN
METHODE PRIVEE
DEBUT
   PROCEDURE produit_achete()
   DEBUT
      quantite <- 0
   FIN
FIN
```

Ainsi, la méthode `produit_achete()` ne pourra pas être appelée en dehors des sous-programmes de la classe `Produit`. L'attribut `nom` de la classe ne pourra pas non plus être modifié dans le programme principal, il ne pourra être affecté que dans les méthodes de la classe.

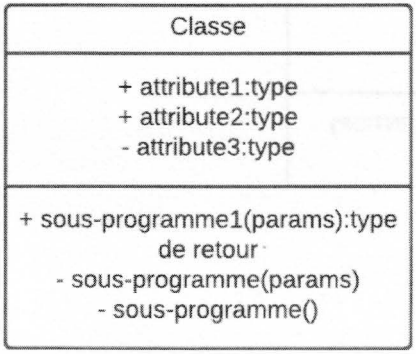
2.5 Un aperçu de l'UML

Notre classe `Produit` illustre le fait que la déclaration algorithmique d'une classe n'est pas lisible rapidement.

Afin de donner un aperçu d'une classe, un langage a été créé : l'**UML** pour *Unified Modeling Language*. Ce langage fournit plusieurs schémas, appelés **diagrammes**, pour représenter tous les aspects d'un programme orientée objet.

Pour notre introduction, nous nous contenterons du **diagramme de classes**.

Un diagramme de classes représente toutes les classes du programme sans l'implémentation des méthodes de ces classes.



Un exemple de classe en UML

Une classe est en effet uniquement représentée par son nom, ses attributs et la **signature** de ses sous-programmes, comme le montre la figure ci-dessus. La signature d'un sous-programme est uniquement la première ligne de sa déclaration (nous nous arrêtons juste avant le bloc `VAR` ou `DEBUT` si le sous-programme ne possède pas de variables locales).

Pour indiquer la visibilité, UML utilise :

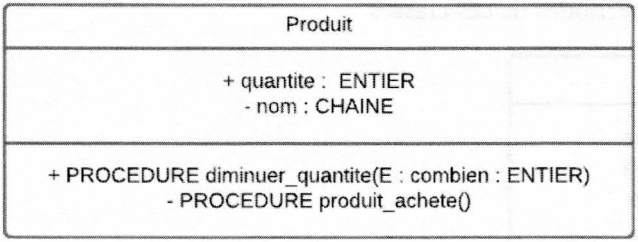
– le "**-**" pour la visibilité privée ;

– le "**+**" pour la visibilité publique.

Dans la classe représentée dans la figure ci-dessus, les deux premiers attributs sont donc publics, le troisième privé. Cette classe ne possède qu'une méthode publique, la première, les deux dernières étant privées.

Pour analyser un algorithme ou programme existant, avec cette représentation schématique, les classes sont plus rapidement lisibles, leur structure algorithmique sert uniquement pour lire l'implémentation des méthodes de la classe.

Modélisons notre classe `Produit` avec l'UML dans la figure ci-dessous pour conclure cette section :



La classe Produit en UML

3. Travailler avec les objets

3.1 Introduction

Les premières sections de ce chapitre ont présenté les principes des classes et des objets. Cette section est dédiée à la manipulation des objets, de leur création jusqu'à leur destruction en mémoire.

Comme le lecteur peut s'en douter, une classe étant une structure de données plus que complexe, un objet sera donc forcément un pointeur.

3.2 Instanciation et allocation mémoire

Pour instancier un objet, nous utiliserons le même opérateur que pour les pointeurs : `NOUVEAU`. Cependant, ce qui suit cet opérateur va être différent, cela ne sera pas le nom de la classe, mais le **constructeur** de la classe.

3.2.1 Constructeur

Le constructeur d'une classe est une méthode **obligatoire** dans la classe. Il permet d'allouer la mémoire de l'objet qui instancie la classe.

Le constructeur a toujours le **même nom** que la classe et ne **retourne rien**. Il peut prendre des paramètres ou non, selon nos besoins.

Par convention, le constructeur prend en paramètres les valeurs à affecter aux attributs de l'objet. Dans le cas d'un constructeur sans paramètre, que nous appelons constructeur par défaut, il initialise les attributs à des valeurs par défaut comme son nom l'indique.

Le constructeur se définit par le mot-clé `CONSTRUCTEUR` en premier dans le bloc des méthodes publiques et ce, en étant la première méthode de ce bloc.

Implémentons donc un constructeur dans notre classe `Produit` pour illustrer cette méthode spécifique.

```
CLASSE Produit
ATTRIBUT PUBLIC
DEBUT
   quantite : ENTIER
FIN
ATTRIBUT PRIVE
DEBUT
   nom : CHAINE
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Produit(name : CHAINE, combien : ENTIER)
   DEBUT
      nom <- name
      quantite <- combien
   FIN
   PROCEDURE diminuer_quantite(E : combien : ENTIER)
   DEBUT
      quantite <- quantite - combien
   FIN
FIN
METHODE PRIVEE
DEBUT
   PROCEDURE produit_achete()
   DEBUT
```

```
        quantite <- 0
    FIN
FIN
PROGRAMME Constructeur_exemple
VAR
    mon_produit : PRODUIT
    nom : CHAINE
    quantite : ENTIER
DEBUT
    ECRIRE("Entrer le nom de votre produit)
    nom <- LIRE
    ECRIRE("Entrer la quantité de votre produit)
    quantite <- LIRE
    mon_produit <- NOUVEAU Produit(nom, quantite)
FIN
```

Le constructeur que nous avons créé permet d'initialiser le nom et la quantité du produit lors de l'initialisation de notre objet `mon_produit`.

Nous remarquons également que l'étoile précédant l'identifiant des pointeurs a disparu car un objet est obligatoirement un pointeur dont la mémoire est allouée grâce à l'appel du constructeur.

3.2.2 Destructeur

Qui dit allocation de mémoire, dit forcément désallocation de mémoire. Pour désallouer la mémoire d'un objet, notamment si la classe déclare des attributs de type pointeur, nous pouvons définir à nouveau une méthode spécifique : le **destructeur**.

Le destructeur est **automatiquement** appelé lorsque l'objet prend comme valeur `NULL` pour l'effacer de la mémoire du programme.

Le destructeur se définit grâce au mot-clé `DESTRUCTEUR` suivi par `~NomDeLaClasse` dans le bloc public des méthodes de la classe, juste après le constructeur. Contrairement au constructeur, il est **facultatif**.

Nous n'avons besoin de définir cette méthode que si notre classe possède des attributs de type pointeur. En effet, pour les attributs non pointeurs, la mémoir sera désallouée automatiquement.

Ajoutons donc un pointeur à notre classe `Produit` pour l'attribut `nom`.

```
CLASSE Produit
ATTRIBUT PUBLIC
DEBUT
   quantite : ENTIER
FIN
ATTRIBUT PRIVE
DEBUT
   *nom <- NULL : CHAINE
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Produit(name : CHAINE, combien : ENTIER)
   DEBUT
      nom <- NOUVEAU CHAINE
      *nom <- name
      quantite <- combien
   FIN
   DESTRUCTEUR ~Produit()
   DEBUT
      nom <- NULL
   FIN

   PROCEDURE diminuer_quantite(E : combien : ENTIER)
   DEBUT
      quantite <- quantite - combien
   FIN
FIN
METHODE PRIVEE
DEBUT
   PROCEDURE produit_achete()
   DEBUT
      quantite <- 0
   FIN
FIN
PROGRAMME Destructeur_exemple
VAR
   mon_produit : PRODUIT
   nom : CHAINE
   quantite : ENTIER
DEBUT
   ECRIRE("Entrer le nom de votre produit)
   nom <- LIRE
   ECRIRE("Entrer la quantité de votre produit)
   quantite <- LIRE
   mon_produit <- NOUVEAU Produit(nom, quantite)
```

```
    // Nous effaçons notre objet de la mémoire
    mon_produit <- NULL
FIN
```

Sans la méthode du destructeur, le pointeur nom serait resté alloué, ce qui aurait provoqué une fuite mémoire. Le destructeur est automatiquement appelé lors de la dernière instruction quand nous affectons NULL à notre objet. Nous pouvons également remarquer que le constructeur sert également à allouer la mémoire des attributs de la classe de type pointeur.

3.3 Appeler les méthodes

Nos objets étant implicitement des pointeurs, nous devons donc utiliser l'opérateur -> pour accéder aux méthodes et attributs publics.

```
CLASSE Produit
ATTRIBUT PUBLIC
DEBUT
   quantite : ENTIER
FIN
ATTRIBUT PRIVE
DEBUT
   *nom <- NULL : CHAINE
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Produit(name : CHAINE, combien : ENTIER)
   DEBUT
      nom <- NOUVEAU CHAINE
      *nom <- name
      quantite <- combien
   FIN
   DESTRUCTEUR ~Produit()
   DEBUT
      nom <- NULL
   FIN

   PROCEDURE diminuer_quantite(E : combien : ENTIER)
   DEBUT
      quantite <- quantite - combien
   FIN
FIN
```

```
METHODE PRIVEE
DEBUT
   PROCEDURE produit_achete()
   DEBUT
      quantite <- 0
   FIN
FIN
PROGRAMME Appel_méthode_exemple
VAR
   mon_produit : PRODUIT
   nom : CHAINE
   quantite : ENTIER
DEBUT
   ECRIRE("Entrer le nom de votre produit)
   nom <- LIRE
   ECRIRE("Entrer la quantité de votre produit)
   quantite <- LIRE
   mon_produit <- NOUVEAU Produit(nom, quantite)
   // Nous appelons la méthodre diminuer_quantite
   mon_produit->diminuer_quantite(1)
   mon_produit <- NULL
FIN
```

Notre algorithme pour modéliser nos produits en programmation orientée objet est maintenant finalisé et propre.

Revenons à nos listes et arbres qui sont des exemples parfaits pour travailler sur les classes.

3.3.1 Listes et composition/agrégation

Dans le chapitre Passons en mode confirmé, nous avons utilisé des structures nœud et liste pour définir notre liste simplement chaînée :

```
STRUCTURE maillon
DEBUT
   valeur : type
   *suivant <- NULL : maillon
FINSTRUCTURE

STRUCTURE liste
DEBUT
   taille <- 0 : ENTIER
   *premier <- NULL : maillon
FINSTRUCTURE
```

Transposons ces structures en classes :

```
Classe Maillon
ATTRIBUT PUBLIC
DEBUT
   valeur : type
   suivant : Maillon
FIN

Classe Liste
ATTRIBUT PUBLIC
DEBUT
   premier : Maillon
FIN
ATTRIBUT PRIVE
DEBUT
   taille : ENTIER
FIN
```

Les champs des structures deviennent les attributs de la classe correspondant à la structure. Nous choisissons de protéger uniquement l'attribut `taille` de la classe `Liste` en lui donnant la visibilité privée. En effet, il paraît logique que seule la liste puisse modifier sa taille.

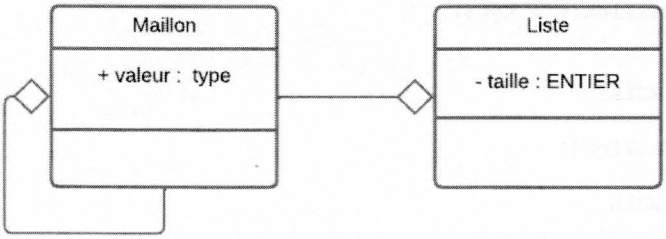
Dans nos classes, il existe un lien entre les classes `Liste` et `Maillon` et un autre entre la classe `Maillon` et elle-même. Ces deux liens représentent les deux liens possibles entre les classes en UML :

- L'**agrégation** : l'objet contenu dans un autre peut vivre sans l'objet le contenant.
- La **composition** : lorsque l'objet conteneur meurt, l'objet contenu meurt également d'office.

Un exemple d'agrégation de la vie de tous les jours peut être la roue d'une voiture. Lorsque la voiture ne fonctionne plus et va à la casse pour être détruite, la roue peut être récupérée pour être réutilisée.

La composition peut être illustrée avec le cerveau d'un être humain. Lorsqu'un être humain décède, son cerveau décède également avec lui.

Comme le montre la figure ci-dessous, l'agrégation est modélisée par un losange vide et la composition par un losange plein. Les losanges sont toujours du côté de la classe qui contient l'autre classe. Avec la représentation de ces liens, les attributs `suivant` et `premier` ne sont plus à écrire dans la liste des attributs des classes en UML. Cependant, ils restent obligatoires à déclarer dans l'algorithme des classes.



Première représentation des listes simplement chaînées en UML

Nous pouvons manipuler nos listes simplement chaînées en ajoutant, insérant ou supprimant un maillon de ces dernières. Les fonctions et procédures correspondantes vont maintenant devenir les méthodes de la classe `Liste`. Nous pouvons remarquer que la classe `Noeud` ne possède pas de méthode, hors son constructeur et son destructeur, et cela est tout à fait permis en programmation orientée objet.

Notre diagramme de classes représentant les listes simplement chaînées devient celui de la figure ci-dessous :

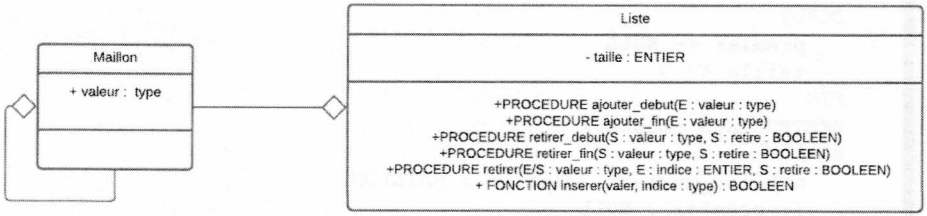


Diagramme de classes pour les listes simplement chaînées

```
Classe Maillon
ATTRIBUT PUBLIC
DEBUT
   valeur : type
   suivant: Maillon
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Maillon(v : type)
   DEBUT
      valeur <- v
      suivant <- NULL
   FIN
   DESTRUCTEUR ~Maillon()
   DEBUT
      suivant <- NULL
   FIN
FIN

Classe Liste
ATTRIBUT PUBLIC
DEBUT
   premier : Maillon
FIN
ATTRIBUT PRIVE
DEBUT
   taille : ENTIER
FIN
METHODE PUBLIC
DEBUT
CONSTRUCTEUR Liste()
   DEBUT
      premier <- NULL
      taille <- 0
   FIN
   DESTRUCTEUR ~Liste()
   VAR
      maillon_courant, maillon : MAILLON
      SI premier ? NULL
      ALORS
         maillon_courant <- premier
         TANTQUE maillon_courant ? NULL
         FAIRE
            maillon <- maillon_courant->suivant
            maillon <- NULL
```

```
            maillon_courant <- maillon_courant->suivant
         FINTANTQUE
      FINSI
   FIN
   PROCEDURE ajouter_debut(E : valeur : type)
   DEBUT
      ...
   FIN
   PROCEDURE ajouter_fin(E : valeur : type)
   DEBUT
      ...
   FIN
   PROCEDURE retirer_debut(S : valeur : type, S : retire : BOOLEEN)
   DEBUT
      ...
   FIN
   PROCEDURE retirer_fin(S : valeur : type, S : retire : BOOLEEN)
   DEBUT
      ...
   FIN
   PROCEDURE retirer(E/S : valeur : type, E : indice : ENTIER,
S : retire : BOOLEEN)
   DEBUT
      ...
   FIN
   FONCTION inserer(valeur: type, indice : ENTIER) : BOOLEEN
   DEBUT
   VAR
      insere <- FAUX : BOOLEEN
      *maillon_avant : Maillon
      *maillon_nouveau : Maillon
      i : ENTIER
   DEBUT
      SI indice > 0 ET taille <= indice ET premier ? NULL
      ALORS
         maillon <- premier
         POUR i ALLANT DE 2 à indice AU PAS DE 1
         FAIRE
            maillon_avant <- maillon.suivant
         FINPOUR
         maillon_nouveau <- NOUVEAU Maillon
         maillon_nouveau-> valeur <- valeur
         maillon_nouveau->suivant <- maillon_avant->suivant
         maillon_avant->suivant <- maillon_nouveau
         insere <- VRAI
```

```
        FINSI
    FIN
FIN
    liste.taille <- liste.taille - 1

        RETOURNER(insere)
    FIN
FIN
```

Les attributs sont maintenant initialisés dans les constructeurs et non plus lors de leur déclaration, ce qui est la convention en programmation orientée objet.

Le destructeur de la classe `Liste` va parcourir la liste en désallouant chaque maillon, ce qui devait être fait à la main dans chaque algorithme utilisant la structure liste avant que l'algorithme ne s'arrête. Nous voyons donc avec cette méthode un premier avantage de la modélisation en classe.

Les méthodes de la classe `Liste` permettent quant à elles d'alléger la liste des paramètres car elles ont un accès direct à la liste, son premier maillon et sa taille. Cela nous permet de transformer la procédure d'insertion en fonction retournant un booléen indiquant si l'insertion a pu être effectuée. Cette méthode est donc plus légère à gérer que l'ancienne procédure.

3.3.2 Arbres binaires en orienté objet

Nous pouvons également transposer notre arbre binaire en programmation orientée objet, comme le montre la figure ci-dessous représentant le diagramme de classes de ce type complexe de donnée :

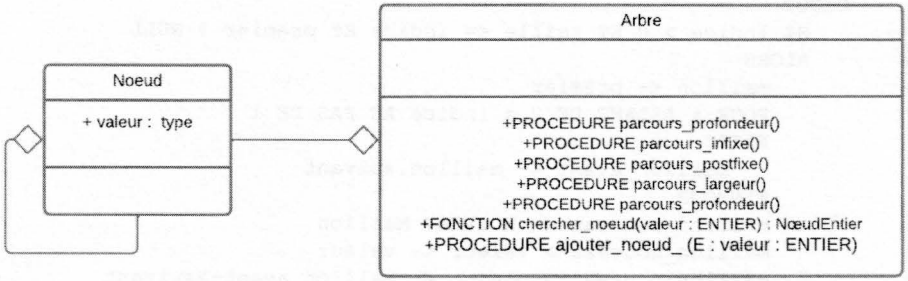


Diagramme de classes pour les arbres binaires

Pour rappel, voici les structures utilisées au chapitre Passons en mode confirmé pour définir un arbre binaire d'entiers.

```
STRUCTURE nœud_entier
DEBUT
   *noeud_gauche <- NULL : maillon
   *noeud_droit  <- NULL : maillon
   valeur : ENTIER
FINSTRUCTURE
STRUCTURE arbre_binaire_entier
DEBUT
   *racine <- NULL : noeud_entier
FINSTRUCTURE
```

Ces deux structures peuvent être traduites avec les classes suivantes :

```
Classe NoeudEntier
ATTRIBUT PUBLIC
DEBUT
   noeud_gauche : NoeudEntier
   noeud_droit  : NoeudEntier
   valeur : ENTIER
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR NoeudEntier(v: *valeur)
   DEBUT
      valeur <- v
      noeud_gauche <- NULL
      noeud_droit <- NULL
   FIN
   DESTRUCTEUR ~NoeudEntier()
   DEBUT
      noeud_gauche <- NULL
      noeud_droit  <- NULL
   FIN
FIN

Classe ArbreBinaireEntier
ATTRIBUT PUBLIC
DEBUT
   Racine : NoeudEntier
FIN
METHODE PUBLIC
```

```
DEBUT
   CONSTRUCTEUR ArbreBinaireEntier()
   DEBUT
      racine <- NULL
   FIN
   DESTRUCTEUR ~ArbreBinaireEntier()
   VAR
      arbre_temporaire : ArbreBinaireEntier
   DEBU
      SI arbre.racine ? NULL
      ALORS
         arbre_temporaire.racine <- racine
   parcours_profondeur_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
      parcours_profondeur_arbre_binaire(arbre_temporaire.
racine->noeud_gauche)
      racine <- NULL
      FINSI
FIN
   PROCEDURE parcours_profondeur()
   DEBUT
      ...
   FIN
   PROCEDURE parcours_infixe ()
   DEBUT
      ...
   FIN
   PROCEDURE parcours_postfixe ()
   DEBUT
      ...
   FIN
   PROCEDURE parcours_largeur ()
   DEBUT
      ...
   FIN
   FONCTION chercher_noeud(valeur : ENTIER) : NœudEntier
   DEBUT
      ...
   FIN
   PROCEDURE ajouter_noeud(E : valeur : ENTIER)
   DEBUT
      ...
   FIN
FIN
```

La logique reste la même que celle des listes simplement chaînées. Les méthodes allègent les paramètres et donc leur code et le destructeur nettoie proprement la mémoire.

3.4 Héritage simple

Outre la réorganisation des données et des liens entre elles, la programmation orientée objet allège encore plus les algorithmes et les programmes grâce à l'héritage.

Pour éviter de copier coller les mêmes attributs et méthodes d'une classe dans une autre, nous pouvons faire hériter cette classe fille d'une classe mère. Ainsi, la classe fille aura tous les attributs et toutes les méthodes de sa mère. Cependant, la classe fille n'aura accès qu'aux attributs et méthodes publics de sa mère car elle n'est pas sa classe mère, elle en hérite juste.

Remarque

Nous appelons aussi la classe mère la super classe et la fille la sous-classe. Les deux appellations sont utilisées par les développeurs et ont la même signification : le lien d'héritage entre les classes.

Illustrons l'héritage avec un exemple simple : les animaux. Chaque animal peut être soit un herbivore, soit un carnivore, soit un omnivore. Quel que soit son type d'alimentation, un animal reste bien un animal. Ainsi, la classe mère sera la classe `Animal` avec trois classes filles `Herbivore`, `Carnivore` et `Omnivore`.

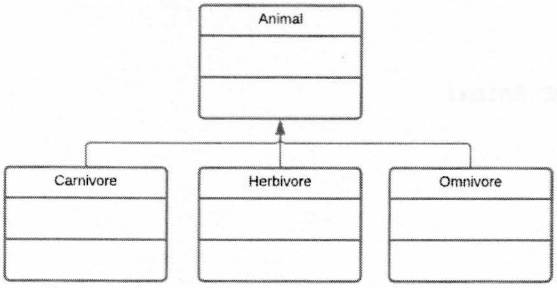


Diagramme de classes pour l'héritage de la classe Animal

Comme illustré dans la figure précédente, le lien d'héritage entre deux classes est une flèche pointant vers la classe mère.

Le diagramme de classes de la figure précédente peut être traduit par l'algorithme suivant où nous utilisons les mots-clés `HERITE DE` pour indiquer le lien d'héritage lors de la déclaration de la classe fille :

```
Classe Animal
ATTRIBUT PUBLIC
DEBUT
   ...
FIN
METHODE PUBLIC
DEBUT
   ...
FIN
Classe Carnivore HERITE DE Animal
ATTRIBUT PUBLIC
DEBUT
   ...
FIN
METHODE PUBLIC
DEBUT
   ...
FIN
Classe Herbivore HERITE DE Animal
ATTRIBUT PUBLIC
DEBUT
   ...
FIN
METHODE PUBLIC
DEBUT
   ...
FIN
Classe Omnivore HERITE DE Animal
ATTRIBUT PUBLIC
DEBUT
   ...
FIN
METHODE PUBLIC
DEBUT
   ...
FIN
```

Reprenons nos listes et plus particulièrement nos listes doublement chaînées. Les maillons de ces listes possèdent juste un attribut de type `maillon` de plus pour aller au maillon précédent. Il s'agit donc bien d'un héritage entre les deux types de maillons, comme le représente le diagramme de classes de la figure ci-dessous :

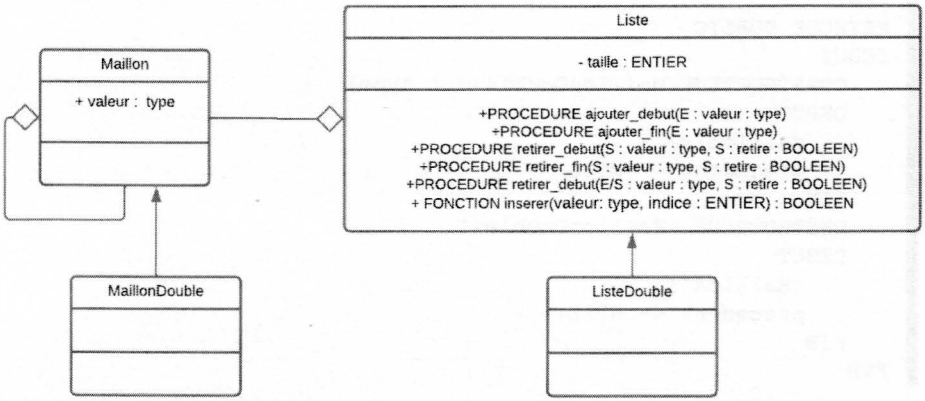


Diagramme de classes des listes avec héritage

En effet, le maillon d'une liste doublement chaînée possède deux maillons : le suivant, comme dans le maillon de la liste simplement chaînée, auquel il ajoute un maillon vers le précédent.

```
Classe Maillon
ATTRIBUT PUBLIC
DEBUT
   valeur : type
   suivant: Maillon
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Maillon(v : type)
   DEBUT
      valeur <- v
      suivant <- NULL
   FIN
   DESTRUCTEUR ~Maillon()
   DEBUT
      suivant <- NULL
```

```
   FIN
FIN
Classe MaillonDouble HERITE DE Maillon
ATTRIBUT PUBLIC
DEBUT
   precedent : MaillonDouble
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR MaillonDouble(v : type)
   DEBUT
      Maillon(v)
      precedent <- NULL
   FIN
   DESTRUCTEUR ~MaillonDouble()
   DEBUT
      ~Maillon()
      precedent <- NULL
   FIN
FIN
```

Grâce à l'héritage, nous pouvons utiliser les méthodes de la classe mère `Maillon` dans la classe fille `MaillonDouble` afin de ne pas réécrire les mêmes instructions (donc éviter un copier/coller inutile). Cette factorisation de code peut se lire à trois endroits de la classe `MaillonDouble` :

- Les attributs `valeur` et `precedent` sont hérités de la classe `Maillon`.
- Le constructeur appelle celui de la classe `Maillon` pour allouer le maillon suivant.
- Le destructeur appelle celui de la classe `Maillon` pour désallouer le maillon suivant.

À la lecture de la classe `MaillonDouble`, vous devez être intrigué par l'attribut `suivant` qui reste de type `Maillon`, ce qui ne semble pas logique. Et pourtant ! Nous allons voir que ce typage ne pose nullement de problème avec la programmation orientée objet dans la prochaine section.

4. Pour aller plus loin

4.1 Polymorphisme

Le polymorphisme est un grand mot pour décrire un comportement tout à fait logique de l'héritage dans la programmation orientée objet. Il amplifie le lien entre une classe mère et ses classes filles, aussi bien au niveau de l'instanciation qu'au niveau de l'implémentation des méthodes.

4.1.1 Objet

Grâce au polymorphisme, un objet de type de la classe mère peut appeler le constructeur d'une de ses classes fille. Pour vous en souvenir, vous pouvez utiliser la technique mnémonique suivante : une mère peut être enceinte d'une fille, donc la contenir, mais une fille ne peut pas être enceinte de sa mère.

Ainsi nous pouvons rendre notre classe `MaillonDouble` adéquate :

```
Classe Maillon
ATTRIBUT PUBLIC
DEBUT
   valeur : type
   suivant: Maillon
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Maillon(v : type)
   DEBUT
      valeur <- v
      suivant <- NULL
   FIN
   DESTRUCTEUR ~Maillon()
   DEBUT
      suivant <- NULL
   FIN
FIN
Classe MaillonDouble HERITE DE Maillon
ATTRIBUT PUBLIC
DEBUT
   precedent : MaillonDouble
FIN
```

```
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR MaillonDouble(v : type)
   DEBUT
      Maillon(v)
      precedent <- NULL
   FIN
   DESTRUCTEUR ~MaillonDouble()
   DEBUT
      ~Maillon()
      precedent <- NULL
   FIN
   PROCEDURE instancier_suivant()
   DEBUT
      suivant <- NOUVEAU MaillonDouble(1)
   FIN
FIN
```

Dans la méthode `instancier_suivant`, nous affectons bien l'objet avec le constructeur de la classe `MaillonDouble`. Vu que le constructeur de la classe `MaillonDouble` demande une valeur en paramètre, nous sommes dans l'obligation de lui en fournir une dans cette instanciation, ce qui au final n'a pas de sens, car l'attribut `suivant` n'a pas besoin de valeur au moment de sa création.

4.1.2 Surcharge de méthodes

Pour éviter l'illogisme de l'initialisation de l'attribut `suivant` dans la classe `MaillonDouble`, nous pouvons utiliser le principe de la **surcharge de méthodes** de la programmation orientée objet.

Dans une même classe, **plusieurs méthodes peuvent avoir le même nom à condition de ne pas avoir la même signature**, c'est-à-dire la même liste de paramètres.

Seul le destructeur d'une classe ne peut pas être surchargé, toutes les autres méthodes, y compris le constructeur, peuvent l'être. En effet, le destructeur ne possède qu'une seule signature possible car il lui est interdit de prendre des paramètres.

Ce principe nous permet donc d'ajouter un nouveau constructeur par défaut dans la classe `MaillonDouble` pour résoudre notre illogisme.

```
Classe Maillon
ATTRIBUT PUBLIC
DEBUT
   valeur : type
   suivant: Maillon
FIN
METHODE PUBLIC
DEBUT
   CONSTRUCTEUR Maillon(v : type)
   DEBUT
      valeur <- v
      suivant <- NULL
   FIN
   DESTRUCTEUR ~Maillon()
   DEBUT
      suivant <- NULL
   FIN
FIN
Classe MaillonDouble HERITE DE Maillon
ATTRIBUT PUBLIC
DEBUT
   precedent : MaillonDouble
FIN
METHODE PUBLIC
DEBUT
CONSTRUCTEUR MaillonDouble(v : type)
   DEBUT
      Maillon(v)
      precedent <- NULL
   FIN
   CONSTRUCTEUR MaillonDouble()
   DEBUT
      suivant <- NULL
      precedent <- NULL
FIN
   DESTRUCTEUR ~MaillonDouble()
   DEBUT
      ~Maillon()
      precedent <- NULL
   FIN
   PROCEDURE instancier_suivant()
   DEBUT
      suivant <- NOUVEAU MaillonDouble()
   FIN
FIN
```

4.1.3 Réécriture de méthodes

Le polymorphisme permet également à une classe fille de **réécrire** l'implémentation des méthodes de sa classe mère sans en changer la signature.

Regardons ce nouveau principe avec l'héritage entre notre classe `Liste` et notre classe `ListeDouble`.

```
Classe Liste
ATTRIBUT PUBLIC
DEBUT
   premier : Maillon
FIN
ATTRIBUT PRIVE
DEBUT
   taille : ENTIER
FIN
METHODE PUBLIC
DEBUT
CONSTRUCTEUR Liste()
   DEBUT
      premier <- NULL
      taille <- 0
   FIN
   DESTRUCTEUR ~Liste()
   VAR
      maillon_courant, maillon : MAILLON
   DEBUT
      SI premier ≠ NULL
      ALORS
         maillon_courant <- premier
         TANTQUE maillon_courant ≠ NULL
         FAIRE
            maillon <- maillon_courant->suivant
            maillon <- NULL
            maillon_courant <- maillon_courant->suivant
         FINTANTQUE
      FINSI
   FIN
   PROCEDURE ajouter_debut(E : valeur : type)
   DEBUT
      ...
   FIN
   PROCEDURE ajouter_fin(E : valeur : type)
   DEBUT
      ...
```

```
    FIN
    PROCEDURE retirer_debut(S : valeur : type, S : retire : BOOLEEN)
    DEBUT
       ...
    FIN
    PROCEDURE retirer_fin(S : valeur : type, S : retire : BOOLEEN)
    DEBUT
       ...
    FIN
    PROCEDURE retirer_debut(E/S : valeur : type, S : retire : BOOLEEN)
    DEBUT
       ...
    FIN
    FONCTION inserer(valeur: type, indice : ENTIER) : BOOLEEN
    DEBUT
    VAR
       insere <- FAUX : BOOLEEN
       *maillon_avant : Maillon
       *maillon_nouveau : Maillon
       i : ENTIER
    DEBUT
       SI indice > 0 ET taille <= indice ET premier ? NULL
       ALORS
          maillon <- premier
          POUR i ALLANT DE 2 à indice AU PAS DE 1
          FAIRE
             maillon_avant <- maillon.suivant
          FINPOUR
          maillon_nouveau <- NOUVEAU Maillon
          maillon_nouveau-> valeur <- valeur
          maillon_nouveau->suivant <- maillon_avant->suivant
          maillon_avant->suivant <- maillon_nouveau
          insere <- VRAI
       FINSI
    FIN
FIN
    liste.taille <- liste.taille - 1

        RETOURNER(insere)
    FIN
FIN

Classe ListeDouble HERITE DE Liste
METHODE PUBLIC
DEBUT
CONSTRUCTEUR ListeDouble()
```

```
    DEBUT
       Liste()
    FIN
    DESTRUCTEUR ~ListeDouble()
       ~ListeDouble
    FIN
    PROCEDURE ajouter_debut(E : valeur : type)
    VAR
       *maillon : MaillonDouble
    DEBUT
       SI premier = NULL
       ALORS
          maillon <- NOUVEAU MaillonDouble(valeur)
          premier <- maillon
       SINON
          maillon <- NOUVEAU MaillonDouble(valeur)
          maillon->suivant <- premier
          premier->precedent <- maillon
          premier <- maillon
       FINSI
    FIN
    PROCEDURE ajouter_fin(E : valeur : type)
    DEBUT
       ...
    FIN
    PROCEDURE retirer_debut(S : valeur : type, S : retire : BOOLEEN)
    DEBUT
       ...
    FIN
    PROCEDURE retirer_fin(S : valeur : type, S : retire : BOOLEEN)
    DEBUT
       ...
    FIN
    PROCEDURE retirer_debut(E/S : valeur : type, S : retire : BOOLEEN)
    DEBUT
       ...
    FIN
       FONCTION inserer(valeur: type, indice : ENTIER) : BOOLEEN
    DEBUT
       ...
    FIN
FIN
```

L'avantage de la réécriture des méthodes est de ne pas avoir besoin de changer la signature des méthodes dans la classe fille, ce qui nous permet d'en garder le sens premier. Pour illustrer ce principe, nous avons juste réécrit la méthode `ajouter_debut` de la liste simplement chaînée dans la classe `ListeDouble`. Vu que les maillons sont de type `MaillonDouble` dans cette classe, toutes les méthodes de la classe `Liste` sont à réécrire pour assurer la gestion doublement chaînée de cette liste.

4.2 Héritage multiple

La plupart des langages de programmation orientée objet ne permettent que l'héritage simple : une classe fille ne peut hériter que d'une seule classe mère.

Cependant, plusieurs langages, comme par exemple le C++ ou encore le Python, implémentent la possibilité qu'une classe fille hérite de plusieurs classes mères, ce qui s'appelle **l'héritage multiple**.

Dans notre exemple précédent d'héritage avec la classe `Animal`, nous pouvons remarquer que les animaux omnivores sont en fait des animaux herbivores et carnivores à la fois. Il s'agit bien d'un héritage multiple comme l'indique le diagramme de classes de la figure ci-après.

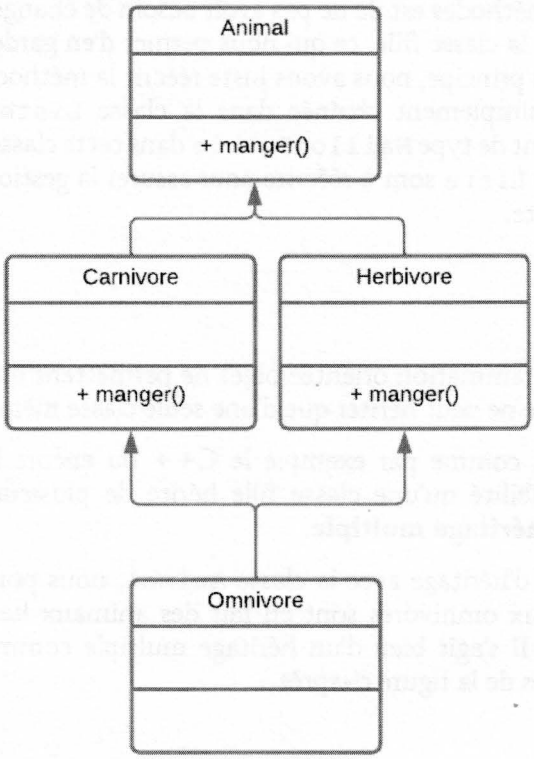


Diagramme de classes d'un héritage multiple en losange

L'héritage multiple peut être complexe à gérer au niveau du polymorphisme, notamment au niveau de la réécriture des méthodes. En effet, si la classe `Animal` définit une méthode `manger()` et que ses deux filles `Carnivore` et `Herbivore` la réécrivent toutes deux, que se passe-t-il si une instanciation de la classe `Omnivore` appelle cette méthode `manger()` sans la redéfinir ?

Est-ce celle de la classe `Carnivore` qui sera appelée ? Celle de la classe `Herbivore` ? Nous ne pouvons pas le deviner, tout comme l'ordinateur d'ailleurs.

Dans ce type spécifique d'héritage multiple, appelé héritage en **losange** ou en **diamant**, il nous faut obligatoirement redéfinir la méthode `manger()` dans la classe `Omnivore` pour lui indiquer si la machine doit appeler la méthode définie dans la classe `Herbivore` ou celle définie dans la classe `Carnivore`, ou les deux, ou encore une nouvelle implémentation.

Nous ne développerons pas davantage l'héritage multiple dans cet ouvrage car nous introduisons juste les grands principes de la programmation orientée objet. Comme indiqué dans la première section, le lecteur voulant approfondir ses connaissances et compétences peut consulter des ouvrages tels que UML 2.5 et Design Patterns ou encore Apprendre la Programmation Orientée Objet avec le langage Python dans la collection Ressources informatiques aux Éditions ENI.

5. L'objet en Python

5.1 Objet et classe en Python

La programmation orientée objet est implémentée dans le langage Python. Les classes sont définies grâce au mot-clé `class`. Pour un code propre, il est demandé de créer **un fichier par classe, et un fichier pour le main** de notre programme.

```
# fichier point.py
class Point :
   pass

# fichier main.py
from point import Point
mon-point…
```

Nous créons donc une classe `Point` dans un fichier *point.py* que nous importons dans le fichier contenant le `main`, le fichier *main.py*.

Pour faciliter l'import de nos classes, les fichiers des scripts doivent tous être dans le **même répertoire**.

Les attributs d'une classe en Python ne peuvent être déclarés sans être initialisés du fait du typage dynamique du langage. Du fait de la syntaxe de ce langage, les attributs ne peuvent être uniquement déclarés dans les méthodes de la classe. Allons donc étudier comment déclarer les méthodes d'une classe pour en déclarer notamment ses attributs..

5.2 Utilisation de self pour les méthodes

En Python, pour différencier une fonction d'une méthode, il nous faut placer le mot-clé `self` en premier paramètre de la méthode, qui est toujours définie par le mot-clé `def`. Ce mot-clé indique que la méthode est attachée à la classe, `self` pouvant être traduit comme "soi-même".

Pour appeler une méthode, nous utilisons l'opérateur "." de l'objet instancié.

Lors de l'appel d'une méthode, ce mot-clé `self` disparait de la liste des paramètres, il n'est obligatoire que lors de sa définition.

Le constructeur est une méthode spéciale qui s'appelle obligatoirement `__init__(self, param1, param2...)` où les paramètres, or le `self`, sont optionnels.

Pour appeler le constructeur et donc instancier un objet, il suffit d'appeler le nom de la classe suivi de la liste des paramètres du constructeur (une paire de parenthèses vide si le constructeur n'est défini qu'avec `self`).

Nous déclarons et initialisons les attributs de la classe dans son constructeur. Pour différencier une variable d'un attribut, un attribut est toujours de la forme `self.identifiant`. Hormis cette contrainte, un attribut se manipule normalement comme une variable ou un objet.

```
# fichier point.py
class Point :
    def __init__(self, x, y):
        # Initialisation et déclaration des attributs
        self.x = x
        self.y = y
    def afficher(self) :
        print("Point :", self.x, "-", self.y)
    def changerCoordonnees(self, nouveaux, nouveauy) :
```

```
        self.x = nouveaux
        self.y = nouveauy

# fichier main.py
from point import Point
point = Point(4,3)
point.afficher()        # affiche en console Point :4 - 3
point.changerCoordonnees(1,2)
point.afficher()        # affiche en console Point :1 - 2
```

Python gérant les allocations mémoire des pointeurs automatiquement, nous ne sommes pas dans l'obligation de définir le destructeur de nos classes. Cependant, pouvons le faire en implémentant la méthode `__del__(self)` si besoin.

```
# fichier point.py
class Point :
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __del__(self):
        print("le point meurt")
    def afficher(self) :
        print("Point :", self.x, "-", self.y)
    def changerCoordonnees(self, nouveaux, nouveauy) :
        self.x = nouveaux
        self.y = nouveauy

# fichier main.py
from point import Point
point = Point(4,3)
```

Lorsque nous exécutons ce script, l'interpréteur Python vide **automatiquement** la mémoire à la fin de cette exécution. Nous aurons donc besoin d'affiché en dernière ligne dans la console le `print` du destructeur : "le point meurt".

5.3 Agrégation et composition

Pour implémenter une relation de composition en Python, il nous faut passer en paramètre du constructeur de la classe maître les paramètres du constructeur de la classe instanciée. Cela nous donne pour une classe `Figure` contenant un point de notre classe `Point` précédente :

```
# composition
form point import Point

class Figure :
   def __init__(self, x, y) :
      self.point = Point(x, y)
```

Avec l'instanciation de la classe `Point` dans le constructeur de la classe `Figure`, l'objet `point` sera désalloué en même temps que l'objet de la classe `Figure` qui le contient.

Pour une relation d'agrégation, nous devons passer en paramètre de la classe conteneur, un objet de la classe contenue.

```
# agrégation
form point import Point

class Figure :
   def __init__(self, point) :
      self.point = point
```

Avec cette instanciation de la classe `Point`, notre objet `point` ne meurt pas en même temps que notre objet instanciant la classe `Figure`. Il est alloué en dehors de la classe, il peut donc continuer à vivre sans aucun objet instanciant la classe `Figure`.

5.4 Héritage simple et polymorphisme

Pour présenter l'héritage simple en Python, nous allons utiliser une classe `Rectangle` et une classe `Carre`.

Un rectangle possède une largeur et une longueur. Il peut s'afficher, calculer son aire et son périmètre. Un carré est un rectangle particulier ayant la même valeur pour sa largeur et sa longueur. Le carré hérite donc du rectangle comme le montre le diagramme de classes de la figure qui suit.

En Python, l'héritage se déclare lors de la déclaration de la classe fille grâce à une paire de **parenthèses** :

```
class Fille(Mere) :
    ...
```

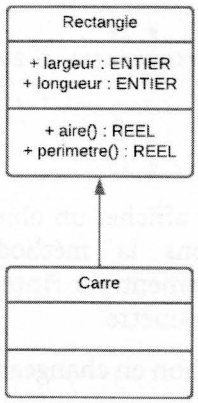


Diagramme de classes pour l'héritage d'un rectangle

```
# fichier rectangle.py
class Rectangle :
    def __init__(self, largeur, longueur) :
        self.largeur = largeur
        self.longueur = longueur
    def perimetre(self) :
        return 2 * (self.largeur + self.longueur)
    def aire(self) :
        return self.largeur * self.longueur
    # Permet d'afficher un objet dans le print de Python
    def __str__(self) :
```

```
        return "Rectangle de largeur "+ str(self.largeur) + " et de
longueur " + str(self.longueur)

# fichier carre.py
from rectangle import Rectangle
class Carre(Rectangle) :
    def __init__(self, largeur) :
        Rectangle.__init__(self, largeur, largeur)
    def __str__(self) :
        return "Carrée de côté "+ str(self.largeur) + "et d'aire " +
str(self.aire())

# fichier main.py
from rectangle import Rectangle
from carre import Carre
if __name__ == '__main__':
    monRectangle = Rectangle(2,8)
    print(monRectangle)
    print("Aire du rectangle :", monRectangle.aire())    # 16
    print("Périmètre du rectangle :", monRectangle.perimetre()) # 20
    monCarre = Carre(3)
    print(monCarre)
    print("Aire du carré :", monCarre.aire())    # 9
    print("Périmètre du carré :", monCarre.perimetre()) # 12
```

Nous utilisons dans ce script une subtilité de Python pour afficher un objet avec la fonction native de Python : nous réécrivons la méthode `__str__(self)`. Cette méthode est appelée automatiquement par l'interpréteur lorsqu'une instruction `print` prend un objet en paramètre.

Cette méthode illustre également le polymorphisme en Python en changeant l'affichage pour la classe `Carre`.

Notons que pour qu'une classe fille puisse appeler une méthode définie dans sa classe mère, il faut préfixer la méthode par `self.`, comme ici pour l'affichage d'un carré qui affiche en plus son aire.

5.5 Pour aller plus loin

5.5.1 Visibilité privée

Par convention, Python demande de laisser les méthodes et les attributs des classes en visibilité publique. Cependant, le langage implémente également une sorte **d'obfuscation** qui permet de rendre les attributs d'une classe privés. Pour ce faire, nous devons déclarer les attributs avec `self.__identifiant`.

```
# fichier rectangleprive.py
class RectanglePrive :
    def __init__(self, largeur, longueur) :
        self.__largeur = largeur
        self.__longueur = longueur
    def perimetre(self) :
        return 2 * (self.__largeur + self.__longueur)
    def aire(self) :
        return self.__largeur * self.__longueur
    # Permet d'afficher un objet dans le print de Python
    def __str__(self) :
        return "Rectangle de largeur "+ str(self.__largeur) + " et
de longueur " + str(self.longueur)
    def getLargeur(self):
        return self.__largeur
    def setLargeur(self, largeur):
        self.__largeur = largeur
    def getLongueur(self):
        return self.__longueur
    def setLargeur(self, longueur):
        self.__longueur = longueur

# fichier carreprive.py
from rectangleprive import RectanglePrive
class CarrePrive(RectanglePrive) :
    def __init__(self, largeur) :
        Rectangle.__init__(self, largeur, largeur)
    def __str__(self) :
        return "Carrée de côté "+ str(getLargeur(self))
```

En rendant privés les attributs de la classe `RectanglePrive`, la classe fille `CarrePrive` n'a plus directement accès à l'attribut `largeur`, elle doit passer par son accesseur avec `self.getLargeur` car cette méthode est définie dans sa classe mère, `RectanglePrive`.

5.5.2 Héritage multiple

Python autorise une classe fille à avoir plusieurs classes mères. Pour ce faire, nous devons lister toutes les mères dans la déclaration de la fille. Reprenons notre héritage multiple avec les animaux et codons-le dans un script Python.

```
class Animal():
    def __init__(self, nom) :
        self.nom = nom
    def mange(self) :
        print(self.nom, "mange")

class Carnivore(Animal):
    def __init__(self, nom) :
        Animal.__init__(self, nom)
    def mange(self) :
        print(self.nom, "mange de la viande")

class Herbivore(Animal):
    def __init__(self, nom) :
        Animal.__init__(self, nom)
    def mange(self) :
        print(self.nom, "mange de l'herbe")

class Omnivore(Carnivore, Herbivore):
    def __init__(self, nom) :
        Carnivore.__init__(self, nom)
        Herbivore.__init__(self, nom)
    def mange(self) :
        print(self.nom, "mange de l'herbe et de la viande")

if __name__ == '__main__':
    hamtaro = Herbivore("hamtaro")
    garfield = Carnivore("garfield")
    pluto = Omnivore("pluto")
    hamtaro.mange()
    garfield.mange()
    pluto.mange()
```

La classe `Omnivore` hérite bien des classes `Carnivore` et `Herbivore`. De ce fait, elle doit respecter deux contraintes pour que le script puisse s'exécuter sans erreur :

- Appeler le constructeur de chacune de ses mères dans son constructeur pour réaliser une allocation mémoire correcte.
- Réécrire la méthode `manger()` car l'interpréteur Python ne peut pas savoir quelle méthode appeler sinon.

5.5.3 Surcharge des opérateurs

Python possède un mécanisme très sympathique pour les opérateurs courants d'algèbre et de comparaison. Nous pouvons en effet les redéfinir pour nos propres classes.

Python définit une méthode particulière pour chaque opérateur du langage. Cette méthode prend en paramètre un objet du type de la classe dans laquelle est définie.

Voici la liste des opérateurs que Python nous permet de redéfinir :

- + avec la méthode `__add__(self, other)` ;
- - avec la méthode `__sub__(self, other)` ;
- * avec la méthode `__mul__(self, other)` ;
- / avec la méthode `__div__(self, other)` ;
- // avec la méthode `__floordiv__(self, other)` ;
- % avec la méthode `__mod__(self, other)` ;
- ** avec la méthode `__pow__(self, other)` ;
- += avec la méthode `__iadd__(self, other)` ;
- -= avec la méthode `__isub__(self, other)` ;
- *= avec la méthode `__imul__(self, other)` ;
- /= avec la méthode `__idiv__(self, other)` ;
- //= avec la méthode `__ifloordiv__(self, other)` ;
- %= avec la méthode `__imod__(self, other)` ;
- **= avec la méthode `__ipow__(self, other)` ;

- < avec la méthode `__lt__(self, other)` ;
- > avec la méthode `__gt__(self, other)` ;
- <= avec la méthode `__le__(self, other)` ;
- >= avec la méthode `__ge__(self, other)` ;
- == avec la méthode `__eq__(self, other)` ;
- != avec la méthode `__ne__(self, other)`.

Ce système nous permet de ne pas définir de méthode pour comparer deux rectangles, nous utiliserons juste l'opérateur de comparaison==.

```
class Rectangle :
    def __init__(self, largeur, longueur) :
        self.largeur = largeur
        self.longueur = longueur
    def perimetre(self) :
        return 2 * (self.largeur + self.longueur)
    def aire(self) :
        return self.largeur * self.longueur
    # Permet d'afficher un objet dans le print de Python
    def __str__(self) :
        return "Rectangle de largeur "+ str(self.largeur) + " et de
longueur " + str(self.longueur)
    def __eq__(self, autreRectangle) :
        return self.largeur == autreRectangle.largeur and
self.longueur == autreRectangle.longueur

if __name__ == '__main__':
    monRectangle = Rectangle(2,8)
    deuxiemeRectangle = Rectangle(8,9)
    print("Les deux rectangles sont-ils égaux ?",
deuxiemeRectangle == monRectangle)
    deuxiemeRectangle.largeur = 2
    deuxiemeRectangle.longueur = 8
    print("Les deux rectangles sont-ils égaux ?",
deuxiemeRectangle == monRectangle)
```

Notre script est d'un seul coup plus lisible pour un autre développeur qui connaît forcément l'opérateur d'égalité.

6. Exercices

6.1 Exercice 1

Modélisez en UML puis avec un algorithme une classe `Duree` possédant trois attributs privés :

- le nombre d'heures de la durée ;
- le nombre de minutes de la durée ;
- le nombre de secondes de la durée.

Et les méthodes publiques suivantes :

- Un constructeur par défaut (tous les attributs auront zéro comme valeur) et un constructeur initialisant toutes les valeurs. On vérifiera que les nombres d'heures, minutes, secondes sont bien positifs, dans le cas contraire on considèrera leurs opposés. On fera également les conversions nécessaires pour que les nombres de minutes et de secondes soient strictement inférieurs à 60.
- Une méthode réalisant l'affichage de la durée (sous la forme 3h10m00s).
- Une méthode convertissant la durée en un nombre de secondes.
- Une méthode rajoutant à la durée un nombre de secondes.

Codez par la suite les scripts Python correspondants.

6.2 Exercice 2

Implémentez dans un algorithme la relation d'héritage entre les classes `Animal`, `Mammifere` et `Oiseau` du diagramme de classes de la figure ci-après.

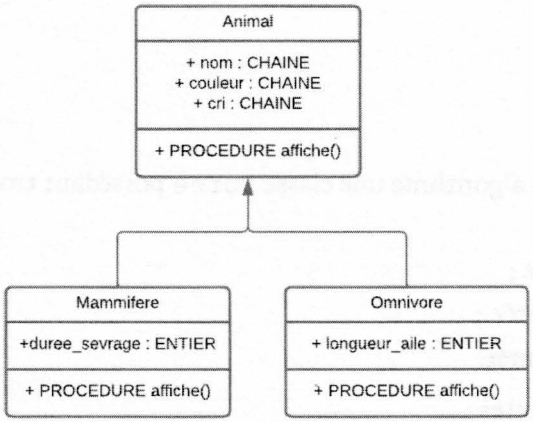


Diagramme de classes des animaux

La procédure `affiche()` effectue un affichage console de tous les attributs de la classe.

Codez les scripts Python correspondants.

6.3 Exercice 3

En Python, codez les classes suivantes :

- Une classe `Cercle`. Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre `rayon`), vous définirez une méthode `surface()`, qui devra renvoyer la surface du cercle.
- Une classe `Cylindre` fille de la classe `Cercle`. Le constructeur de cette nouvelle classe comportera les deux paramètres `rayon` et `hauteur`. Vous y ajouterez une méthode `volume()` qui devra renvoyer le volume du cylindre (rappel : volume d'un cylindre = surface de section x hauteur).
- Une classe `Cone` enfant de la classe `Cylindre`, et dont le constructeur comportera lui aussi les deux paramètres `rayon` et `hauteur`. Cette nouvelle classe possédera sa propre méthode `volume()`, laquelle devra renvoyer le volume du cône (rappel : volume d'un cône = volume du cylindre correspondant divisé par 3).

6.4 Exercice 4

Codez un script Python pour implémenter votre propre liste simplement chaînée contenant les procédures d'ajout et de suppression en début et fin de liste avec une classe. Pour cet exercice, vous devez savoir que le `NULL` se traduit en Python par None.

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

V

W

Ressourcesinformatiques

Algorithmique

Techniques fondamentales de programmation (exemples en Python)

Ce livre sur **l'algorithmique** s'adresse à toute personne désireuse de maîtriser les **bases essentielles de la programmation**. Pour apprendre à programmer, il faut d'abord comprendre ce qu'est vraiment un ordinateur, comment il fonctionne et surtout comment il peut faire fonctionner des programmes, comment il manipule et stocke les données et les instructions, quelle est sa logique. Alors, au fur et à mesure, le reste devient évidence : **variables**, **tests**, **conditions**, **boucles**, **tableaux**, **fonctions**, **fichiers**, jusqu'aux notions avancées comme les **compréhensions de listes** et les **objets**.

Le **langage algorithmique** (ou la syntaxe du pseudo-code des algorithmes) reprend celui couramment utilisé dans les **écoles d'informatique** et dans les formations comme les **BTS**, **DUT**, **première année d'ingénierie** à qui ce livre est principalement destiné et conseillé. Une fois les notions de base acquises, le lecteur trouvera dans ce livre de quoi évoluer vers des notions plus avancées : un chapitre sur **les objets** ouvre les portes de la programmation dans des langages évolués et puissants comme **le C**, **le C++**, **le Java**, **le C# et surtout Python**.

À la fin de chaque chapitre, l'auteur propose de **nombreux exercices corrigés** permettant de consolider ses acquis.

Tous les algorithmes de ce livre sont réécrits en **Python** et les sources, directement utilisables, sont disponibles en téléchargement sur le site www.editions-eni.fr.

Ludivine CREPIN

Docteur en Intelligence Artificielle depuis 2009, **Ludivine CREPIN** est depuis consultante indépendante pour des entreprises au niveau européen, de la start-up à la multinationale. Forte de son expertise, elle propose à ses clients ses services de conseil, de développement et de recherche appliquée pour tous les types de projets informatiques. Également formatrice, elle fait profiter le lecteur de toute sa pédagogie pour l'apprentissage de l'algorithmique basée sur le langage Python.

sur **www.editions-eni.fr** :
➜ Le code source en Python de la plupart des exemples du livre.

Pour plus d'informations :

32 €

ISBN : 978-2-409-04184-6

9 782409 041846

eni

www.editions-eni.fr