



Initiation à l'algorithmique

— Démarche MRV —

Méthode – Résultat – Vérification

JACQUES TISSEAU, ALEXIS NÉDÉLEC, MARC PARENTHOËN

ENIB – Technopôle Brest Iroise
CS 73862 – 29238 Brest cedex 3 – France
www.enib.fr

Table des matières

I	Introduction générale	7
1	Démarche MRV	9
1.1	Introduction	9
1.2	Exemples	9
1.3	Retours d'expériences	13
1.4	Généralisation	15
1.5	Mise en œuvre	16
1.6	Conclusion	19
2	Premiers pas	21
2.1	Organisation du document	21
2.2	Calculatrice PYTHON	21
II	Instructions de base	23
3	Affectation	25
3.1	Rappels de cours	25
3.2	Vie courante : prix d'un livre	28
3.3	Jeux : lancer de dés	30
3.4	Textes : longueurs de chaînes de caractères	32
3.5	Nombres : quotient et reste	34
3.6	Figures : distance entre deux points	36
3.7	Mathématiques : produit scalaire	38
3.8	Physique : conversion d'unités	39
3.9	Informatique : préfixes binaires	41
3.10	Retours d'expériences	43
4	Tests	45
4.1	Rappels de cours	45
4.2	Vie courante : mention au baccalauréat	47
4.3	Jeux : 421	48
4.4	Textes :	48
4.5	Nombres :	49
4.6	Figures :	49
4.7	Mathématiques : graphe de fonction	50
4.8	Physique : états de l'eau	53
4.9	Informatique :	53

4.10 Retours d'expériences	54
5 Boucles	55
5.1 Rappels de cours	55
5.2 Vie courante : dépiler des assiettes	60
5.3 Jeux : rechercher une carte	60
5.4 Textes : compter les voyelles	61
5.5 Nombres : conversion décimal \rightarrow binaire	61
5.6 Figures : tracé d'un heptagone régulier	62
5.7 Mathématiques : intégration de $\cos(x)$	62
5.8 Physique : sorties d'un circuit logique	63
5.9 Informatique :	63
5.10 Retours d'expériences	64
6 Instructions imbriquées	65
6.1 Rappels de cours	65
6.2 Vie courante	65
6.3 Jeux : drapeau tricolore	65
6.4 Textes : recherche d'un motif	66
6.5 Nombres : crible d'Eratostène	66
6.6 Figures :	67
6.7 Mathématiques : zéro d'une fonction	67
6.8 Physique :	68
6.9 Informatique :	68
6.10 Retours d'expériences	69
III Fonctions	71
7 Spécification	73
8 Appels de fonctions	75
9 Récursivité	77
IV Tout en un	79
10 Vie courante : rechercher son chemin	81
11 Jeux :	83
12 Textes : cryptographie	85
13 Nombres :	87
14 Figures : construction de figures	89
15 Mathématiques : systèmes linéaires	91
16 Physique :	93



TABLE DES MATIÈRES

5/95

17 Informatique : machine de Turing

95



TABLE DES MATIÈRES

6/95

Première partie

Introduction générale

Chapitre 1

Démarche MRV

1.1 Introduction

L'objectif de ce document est de présenter la démarche suivie pour répondre aux exercices qui accompagnent le cours d'« **Initiation à l'algorithmique** » du semestre S1 à l'Ecole nationale d'ingénieurs de Brest (ENIB). Cette démarche, dite MRV (Méthode-Résultat-Vérification), est structurée en 3 étapes :

1. on commence par expliciter la méthode générique qui permet de résoudre des problèmes équivalents à celui qui est posé (étape M comme Méthode) ;
2. on applique ensuite cette méthode générique au cas particulier de l'énoncé pour obtenir le résultat attendu par l'exercice (étape R comme Résultat) ;
3. enfin, on réalise une vérification du résultat obtenu à l'aide de techniques alternatives ou complémentaires connues (étape V comme Vérification).

Pour fixer les idées, nous illustrons d'abord la démarche MRV par deux exemples « simples » (section 1.2) : le passage des nœuds « marins » en kilomètres par heure « terrestres » (1.2.1) et le calcul d'une fonction composée de segments de droite (1.2.2). Puis nous commentons quelques retours d'expériences (section 1.3) qui nous ont guidés dans l'élaboration de la démarche MRV : ne pas se tromper d'objectif (1.3.1), expliciter l'implicite (1.3.2) et encourager la rédaction (1.3.3). Forts de cette expérience, nous précisons ensuite les attendus à chaque étape de la démarche MRV (section 1.4) : l'explicitation de la méthode (1.4.1), l'application de la méthode (1.4.2) et la vérification du résultat (1.4.3). Enfin, nous détaillons sa mise en œuvre (section 1.5) à l'ENIB dans le cadre du cours « Initiation à l'algorithmique » (1.5.1) où elle conduit à une triple évaluation des exercices selon une notation adaptée (1.5.2) dans le cadre d'un contrôle continu systématique (1.5.3).

1.2 Exemples

Les exemples décrits dans cette section sont extraits des « **Questionnements de cours** » qui complètent le cours d'Informatique S1 à l'ENIB. Chaque exemple est structuré ici en cinq paragraphes :

1. l'objectif thématique de l'exercice,
2. l'énoncé de l'exercice,
3. la méthode générique utilisée pour résoudre une famille de problèmes équivalents à celui de l'exercice proposé,

4. le résultat obtenu en appliquant la méthode générique au cas particulier de l'énoncé,
5. la vérification du résultat.

1.2.1 Des nœuds aux kilomètres par heure

Objectif Mettre en œuvre l'instruction d'affectation.

Enoncé On veut convertir une certaine quantité n_1 de vitesse exprimée en nœuds (miles nautiques par heure) en la quantité équivalente n_2 exprimée en kilomètres par heure (km/h). Proposer une instruction de type « affectation » qui réalise cette conversion.

Méthode On cherche ici à convertir $n_1 \cdot u_1$ en $n_2 \cdot u_2$ où u_1 et u_2 sont des unités physiques compatibles qui dérivent de la même unité de base u_b du **Système international d'unités**.

$$\begin{cases} u_1 = a_1 \cdot u_b \\ u_2 = a_2 \cdot u_b \end{cases} \Rightarrow \begin{cases} n_1 \cdot u_1 = n_1 \cdot (a_1 \cdot u_b) = (n_1 \cdot a_1) \cdot u_b \\ n_2 \cdot u_2 = n_2 \cdot (a_2 \cdot u_b) = (n_2 \cdot a_2) \cdot u_b \end{cases} \Rightarrow \frac{n_1 \cdot u_1}{n_2 \cdot u_2} = \frac{n_1 \cdot a_1}{n_2 \cdot a_2}$$

Comme on cherche n_2 tel que $n_1 \cdot u_1 = n_2 \cdot u_2$, on a donc :

$$\frac{n_1 \cdot u_1}{n_2 \cdot u_2} = \frac{n_1 \cdot a_1}{n_2 \cdot a_2} = 1 \Rightarrow n_2 = n_1 \cdot \frac{a_1}{a_2}$$

où les coefficients a_i sont documentés dans le Système international d'unités par le **Bureau international des poids et mesures**.

Une fois connus les coefficients a_i , on détermine la quantité n_2 de l'unité u_2 par une affectation simple : $n_2 = n_1 \cdot a_1 / a_2$.

Résultat On applique la méthode précédente à la conversion proposée dans l'énoncé où u_1 représente les nœuds (miles nautiques par heure), u_2 les kilomètres par heure (km/h) et u_b les mètres par seconde (m/s). Le Système international d'unités fournit par ailleurs les facteurs de conversion a_1 (nd \rightarrow m/s) et a_2 (km/h \rightarrow m/s) : $a_1 = 1852/3600$ et $a_2 = 1000/3600$.

Compte-tenu de ces valeurs, le code ci-contre permet de calculer le nombre n_2 de kilomètres par heure en fonction du nombre n_1 de nœuds.

```

1 a1 = 1852/3600
2 a2 = 1000/3600
3 n2 = n1*a1/a2

```

Remarque : on n'a pas cherché à effectuer « à la main » les calculs numériques : PYTHON les fera mieux que nous ; et surtout, on n'a pas cherché non plus à particulariser la 3^{ème} ligne du code en $n_2 = n_1 \cdot 1852 / 1000$: la forme plus abstraite $n_2 = n_1 \cdot a_1 / a_2$ restera identique pour convertir des parsecs en années-lumière (longueurs), des gallons en barils (volumes) ou encore des électron-volts en frigories (énergies), seules les valeurs des coefficients a_i changeront (lignes 1 et 2 du code).

Vérification Pour tester le résultat précédent, on peut comparer les valeurs obtenues par le calcul avec celles de quelques valeurs caractéristiques facilement évaluables « à la main » (exemples : $n_1 = 1$ nd $\Rightarrow n_2 = 1.852$ km/h ou $n_1 = 1/1852$ nd $\Rightarrow n_2 = 1/1000$ km/h).

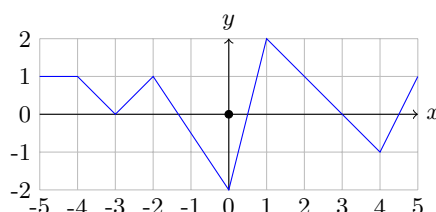
<pre>>>> n1 = 1</pre>	<pre>>>> n1 = 1/1852</pre>
<pre>>>> a1, a2 = 1852/3600, 1000/3600</pre>	<pre>>>> a1, a2 = 1852/3600, 1000/3600</pre>
<pre>>>> n2 = n1*a1/a2</pre>	<pre>>>> n2 = n1*a1/a2</pre>
<pre>>>> n2</pre>	<pre>>>> n2</pre>
<pre>1.852</pre>	<pre>0.0010000000000000002</pre>

On obtient bien par le calcul les résultats escomptés.

1.2.2 Graphe d'une fonction continue affine par morceaux

Objectif Mettre en œuvre l'instruction d'alternative multiple.

Enoncé On considère dans \mathbb{R} la fonction continue f , affine par morceaux, définie sur $[-5; 5]$ par le graphe ci-dessous et $\forall x < -5, f(x) = f(-5)$ et $\forall x > 5, f(x) = f(5)$.



Proposer une instruction de type « alternative multiple » qui calcule la fonction $y = f(x) \forall x \in \mathbb{R}$.

Méthode Il s'agit de déterminer la valeur $y = f(x)$ d'une fonction continue affine par morceaux sur \mathbb{R} . L'axe des réels $] -\infty, x_1, x_2, \dots, x_n, +\infty[$ est donc vu comme une succession d'intervalles $] -\infty, x_1[, [x_1, x_2[, \dots, [x_{n-1}, x_n[$ et $[x_n, +\infty[$ sur lesquels la fonction f est définie respectivement par les fonctions f_1, f_2, \dots, f_n et f_{n+1} :

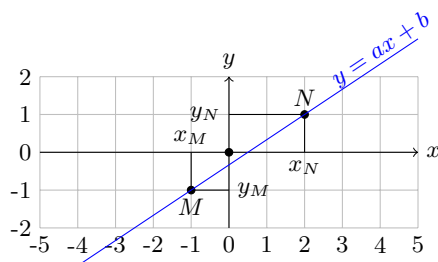
$$\begin{aligned}
 y = f(x) &= f_1(x) & \forall x \in] -\infty, x_1[\\
 &= f_2(x) & \forall x \in [x_1, x_2[\\
 &= f_3(x) & \forall x \in [x_2, x_3[\\
 &= \dots \\
 &= f_n(x) & \forall x \in [x_{n-1}, x_n[\\
 &= f_{n+1}(x) & \forall x \in [x_n, +\infty[
 \end{aligned}$$

Chacune des fonctions f_i correspond à une droite d'équation $y = a_i x + b_i$ où a_i représente la pente de la droite et b_i son ordonnée à l'origine. Lorsqu'on connaît 2 points $M(x_M, y_M)$ et $N(x_N, y_N)$ d'une droite d'équation $y = ax + b$, les coefficients a (pente de la droite) et b (ordonnée à l'origine) de la droite sont obtenus par résolution du système de 2 équations : $y_M = ax_M + b$ et $y_N = ax_N + b$. On obtient alors a et b :

$$a = \frac{y_N - y_M}{x_N - x_M} \text{ et } b = \frac{y_M x_N - y_N x_M}{x_N - x_M}$$

Pour la droite ci-contre :

$$a = \frac{1 - (-1)}{2 - (-1)} = \frac{2}{3} \text{ et } b = \frac{(-1) \cdot 2 - 1 \cdot (-1)}{2 - (-1)} = -\frac{1}{3}$$



On vérifie graphiquement ces résultats : pour passer de M à N , on se déplace de $\Delta x = 3$ horizontalement puis de $\Delta y = 2$ verticalement (d'où la pente $a = \Delta y / \Delta x = 2/3$), et la droite coupe bien l'axe des ordonnées en $y = -1/3$.

Une fois déterminés les coefficients a_i et b_i de chaque droite, on détermine la valeur de la fonction

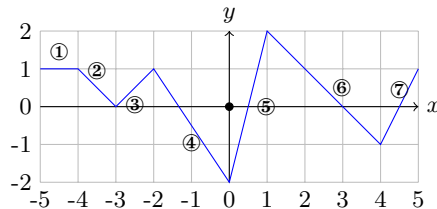
$y = f(x)$ par une alternative multiple du genre :

```

if x < x1 : y = a1*x + b1
elif x < x2 : y = a2*x + b2
elif x < x3 : y = a3*x + b3
...
elif x < xn : y = an*x + bn
else : y = an+1*x + bn+1

```

Résultat On applique la méthode précédente à la fonction f de l'énoncé. Il faut donc déterminer les équations de droite correspondant aux différents segments du graphe de la fonction, à savoir :



- ① $y = 1$
- ② $y = -x - 3$
- ③ $y = x + 3$
- ④ $y = -3x/2 - 2$
- ⑤ $y = 4x - 2$
- ⑥ $y = -x + 3$
- ⑦ $y = 2x - 9$

Compte-tenu de ces équations, le code ci-contre permet de calculer $y = f(x)$, y compris pour $x < -5$ ($y = f(-5) = 1$) et $x > 5$ ($y = f(5) = 1$).

Remarque : on aurait pu simplifier les deux premières lignes de ce code en

```

if x < -4 : y = 1

```

car les instructions associées sont identiques (ie. les fonctions affines sont identiques sur $] -\infty, -5[$ et $[-5, -4[$).

```

1 if x < -5 : y = 1
2 elif x < -4 : y = 1
3 elif x < -3 : y = -x - 3
4 elif x < -2 : y = x + 3
5 elif x < 0 : y = -3*x/2 - 2
6 elif x < 1 : y = 4*x - 2
7 elif x < 4 : y = -x + 3
8 elif x < 5 : y = 2*x - 9
9 else : y = 1

```

Vérification Pour tester le résultat précédent, on peut comparer les valeurs obtenues par le calcul avec celles lues directement sur le graphe pour quelques points caractéristiques. Ces points de mesure sont choisis judicieusement : ils ne correspondent pas aux bornes des intervalles déjà prises en compte dans la méthode mais plutôt à des points où la fonction s'annule (exemples : $x = -4/3$, $1/2$, 3 ou $9/2$) ou à des points d'abscisses aux nœuds de la grille de lecture (exemples : $x = -1$ ou $x = 2$). On peut vérifier par exemple pour $x = -1$ ($y = f(-1) = -1/2$) et $x = 3$ ($y = f(3) = 0$).

```

>>> x = -1
>>> if x < -4 : y = 1
elif x < -3 : y = -x - 3
elif x < -2 : y = x + 3
elif x < 0 : y = -3*x/2 - 2
elif x < 1 : y = 4*x - 2
elif x < 4 : y = -x + 3
elif x < 5 : y = 2*x - 9
else : y = 1

```

```

>>> y
-0.5

```

```

>>> x = 3
>>> if x < -4 : y = 1
elif x < -3 : y = -x - 3
elif x < -2 : y = x + 3
elif x < 0 : y = -3*x/2 - 2
elif x < 1 : y = 4*x - 2
elif x < 4 : y = -x + 3
elif x < 5 : y = 2*x - 9
else : y = 1

```

```

>>> y
0

```

On obtient bien par le calcul les résultats lus sur la grille.

1.3 Retours d'expériences

Les exercices présentés précédemment (section 1.2) ont été proposés à de nombreuses générations d'étudiants de l'ENIB bien avant d'utiliser la démarche MRV. Les retours d'expériences associés ont permis de mettre en évidence au moins trois réflexions méthodologiques : ne pas se tromper d'objectif (1.3.1), expliciter l'implicite (1.3.2) et encourager la rédaction (1.3.3).

1.3.1 Ne pas se tromper d'objectif

Les étudiants de l'ENIB sont issus des voies scientifique (BAC S) et technique (BAC STI) de l'enseignement secondaire. C'est pourquoi, pour donner du sens aux exercices d'algorithmique, de nombreux exemples sont empruntés aux mathématiques, à la physique et plus généralement aux sciences de l'ingénieur. C'est ainsi le cas des deux exemples précédents : la physique pour les conversions d'unités (exercice 1.2.1) et les mathématiques pour les fonctions continues affines par morceaux (exercice 1.2.2).

Ces emprunts interdisciplinaires sont absolument nécessaires pour participer au décloisonnement des disciplines... que les étudiants ont progressivement appris à cloisonner et à isoler au cours de leur scolarité. Mais ils posent le problème de l'objectif thématique de l'exercice. En effet, les étudiants peuvent être si perturbés par la thématique secondaire (ici les mathématiques ou la physique) qu'ils en oublient la thématique principale (ici l'algorithmique), ce qui n'est évidemment pas le but recherché par l'exercice d'algorithmique.

Dans l'exemple de la fonction continue affine par morceaux, un point de blocage souvent rencontré porte sur la détermination de l'équation d'une droite. A ce niveau (premier semestre de l'enseignement supérieur), très peu d'étudiants en difficulté avec l'équation d'une droite, « oseront » écrire quelque chose comme : « supposons que l'on connaisse l'équation de la droite $y = f_i(x)$ dans l'intervalle i considéré, alors l'alternative multiple recherchée s'écrira sous la forme... ». Or, c'est pourtant fondamentalement ce qu'attend l'informaticien, quelle que soit sa « déception » face à la non-maîtrise d'une notion mathématique aussi élémentaire que l'équation d'une droite. Dans une initiation à l'algorithmique, on s'attachera à vérifier la cohérence logique des différentes conditions de l'alternative multiple plutôt que la précision des instructions associées à ces conditions. Ainsi, une réponse cohérente du point de vue des conditions de l'alternative multiple sera beaucoup plus proche de l'objectif recherché qu'une réponse incohérente sur les conditions même avec des équations de droite correctes.

On s'attachera alors à expliciter l'objectif thématique de l'exercice et à renseigner au mieux les éléments nécessaires aux thématiques secondaires. On rappellera aux étudiants que l'évaluation porte sur l'objectif thématique principal et non sur les thématiques secondaires.

1.3.2 Expliciter l'implicite

Si on retient que la qualité d'une réponse est son aptitude à satisfaire strictement aux besoins exprimés dans la question, alors les étudiants de l'ENIB sont devenus des professionnels de la qualité. Dans l'exemple de la conversion d'unités, il est fréquent que la réponse des étudiants tienne en une seule ligne : $n_2 = 1.852 \cdot n_1$, ce qui est effectivement la bonne réponse qui mérite donc, dans leur esprit, la meilleure appréciation. Mais si l'on peut se contenter de cette réponse dans un contexte de physique élémentaire, il n'en va pas de même dans un contexte de formation d'ingénieur.

En algorithmique, on cherche à caractériser différentes propriétés d'un algorithme telles que sa validité, sa réutilisabilité, sa robustesse, sa complexité ou encore son efficacité. Il faut donc

progressivement développer chez l'informaticien débutant des réflexes de validation, de réutilisation, de protection, d'évaluation ou encore d'adaptation au support matériel. L'acquisition de ces réflexes méthodologiques doit être évaluée au même titre que l'acquisition des connaissances proprement dites comme l'affectation ou l'alternative multiple. Il faut donc expliciter auprès des étudiants ces objectifs méthodologiques et ne pas les cantonner au niveau d'objectifs implicites plus ou moins pris en compte dans l'évaluation d'une réponse à un exercice donné.

En premier lieu, il faut s'assurer que l'algorithme est valide : réalise-t-il exactement la tâche pour laquelle il a été conçu ? On demandera alors explicitement aux étudiants de proposer une démarche de vérification de leurs réponses. Dans l'exemple de la fonction continue affine par morceaux, on compare le résultat du calcul par algorithme à une lecture directe sur le graphe pour quelques points caractéristiques bien choisis (points où la fonction s'annule, points aux nœuds de la grille de lecture). Cette méthode ne permet évidemment pas de valider l'algorithme $\forall x \in \mathbb{R}$, mais permet d'invalider l'algorithme proposé s'il existe une incohérence entre le calcul et la lecture directe sur le graphe pour un des points caractéristiques considérés. Cette « vérification » n'en demeure pas moins satisfaisante, et essentielle, dans un cours d'initiation à l'algorithmique.

Dans un deuxième temps, on s'intéresse à sa généralité : l'algorithme est-il réutilisable pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu ? Dans l'exemple de la conversion d'unités, on préfère l'affectation générique ($n2 = n1 \cdot a1/a2$) à l'affectation particulière ($n2 = n1 \cdot 1852/1000$) et encore plus à l'affectation pré-calculée ($n2 = 1.852 \cdot n1$). L'affectation générique s'applique en effet à toute conversion d'unités linéairement dépendantes l'une de l'autre. L'affectation particulière répond correctement mais de manière *ad hoc* au problème posé, ce qui ne permet pas sa réutilisabilité à d'autres types d'unités, y compris même à d'autres unités de vitesse telle que la conversion de miles terrestres par heure en kilomètres par heure ($1 \text{ mi} = 1.609344 \cdot 10^3 \text{ m}$). Quant à l'affectation pré-calculée, outre le fait qu'on n'est jamais à l'abri d'une erreur de calcul, elle ne permettra pas de « remonter » aussi facilement que les deux autres à la signification du coefficient numérique calculé et donc, ne facilitera pas la maintenance du code proposé.

Dans un troisième temps, on s'attachera à le rendre robuste : l'algorithme est-il protégé de conditions anormales d'utilisation ? Dans l'exemple de la conversion d'unités, l'affectation générique $n2 = n1 \cdot a1/a2$ ne s'applique qu'à des unités linéairement dépendantes l'une de l'autre ($u_2 = \alpha u_1$). Il existe cependant des grandeurs qui sont en relation affine l'une de l'autre ($u_2 = \alpha u_1 + \beta$). C'est le cas des températures FARENHEIT t_F et CELSIUS t_C qui sont reliées à la température thermodynamique KELVIN T_K (unité de base des températures) par les relations $t_C = T_K - 273.15$ et $t_F = 9T_K/5 - 459.67$, soit $t_C = 5/9 \cdot (t_F + 459.67) - 273.15$ entre elles. En algorithmique, l'étude des fonctions et de leurs préconditions (conditions d'application) permettra d'aborder systématiquement cette propriété de robustesse.

En ce qui concerne les propriétés d'un algorithme telles que la complexité (combien d'instructions élémentaires seront exécutées pour réaliser la tâche pour laquelle l'algorithme a été conçu ?) et l'efficacité (l'algorithme utilise-t-il de manière optimale les ressources du matériel qui l'exécute ?), elles seront abordées au travers d'exemples précis, leur étude systématique ne relevant pas du cours d'initiation à l'algorithmique de l'ENIB.

Dans tous les cas, on s'attachera à préciser le ou les objectifs méthodologiques de l'exercice qui seront alors évalués explicitement au même titre que l'objectif thématique.

1.3.3 Encourager la rédaction

La rédaction « en bon français » de la réponse à un exercice ne constitue pas le point fort des jeunes étudiants de l'ENIB. Les réponses sont (trop) souvent libellées « simplement » sous

forme de valeurs, de formules, de diagrammes ou de codes informatiques : aucune explication « en bon français » ne vient compléter ni expliciter la réponse, ni la démarche qui a conduit à cette réponse, encore moins la critique de la solution proposée. Et pourtant, si la réponse à la question est attendue, les éléments de discours qui l'accompagnent le sont tout autant, d'autant plus dans une formation d'ingénieurs qui vise à former des professionnels qui devront rédiger des cahiers des charges, des spécifications et des conceptions détaillées, des recettes de tests, des notes de synthèse, écrites comme orales, ou encore des réponses à des appels d'offres. Le métier d'ingénieur ne peut se contenter d'une valeur, d'une formule, d'un diagramme ou d'un code : s'il faut être capable de trouver une solution à un problème donné, il faut aussi savoir « défendre » rationnellement la solution proposée. L'argumentaire qui accompagne la solution proposée doit permettre de mieux comprendre cette solution et augmenter ainsi la confiance du « lecteur » dans les compétences du « rédacteur » à résoudre le problème posé.

On s'attachera alors à prendre en compte explicitement la rédaction dans l'évaluation de la réponse. Il faudra sans doute pour cela, soit augmenter le temps accordé à l'exercice, soit diminuer le nombre d'exercices à résoudre dans un temps imparti, pour permettre à l'étudiant « rédacteur » de soigner cet aspect important de sa réponse.

1.4 Généralisation

A l'aune des exemples précédents (section 1.2) et des retours d'expériences associés (section 1.3), cette section reconsidère les trois étapes de la démarche MRV : explicitation de la méthode (1.4.1), application de la méthode (1.4.2) et vérification du résultat (1.4.3).

1.4.1 Explicitation de la méthode

Etant donné un énoncé qui propose de résoudre un exercice portant sur un cas particulier donné, la première étape de la démarche MRV consiste à décrire une méthode générique qui, lorsqu'on l'appliquera, permettra de résoudre le cas particulier considéré ainsi que tout problème équivalent à celui qui est posé.

Pour un débutant, cette étape d'explicitation d'une méthode générique est une étape difficile. Elle nécessite de développer des capacités d'abstraction qui mettent en œuvre des mécanismes d'induction pour favoriser le passage de données particulières à des propositions plus générales. L'induction est en effet un type de raisonnement qui permet de « remonter » de cas particuliers à la loi qui les régit, des effets à la cause ou encore des conséquences au principe.

C'est également une étape difficile parce que la description d'une méthode peut difficilement se résumer à une valeur, une formule, un diagramme ou un code informatique. Elle nécessite une phase rédactionnelle rigoureuse et suffisamment détaillée pour qu'un lecteur averti puisse appliquer sans hésiter la méthode décrite.

Cette étape permet ainsi de développer des capacités d'abstraction et des capacités rédactionnelles absolument nécessaires aux futurs ingénieurs.

1.4.2 Application de la méthode

Etant donné une méthode générique pour résoudre un ensemble de problèmes équivalents, le deuxième étape de la démarche MRV consiste à appliquer cette méthode à un problème particulier.

Pour un débutant, cette étape d'application d'une méthode générique est une étape assez facile. Elle nécessite de développer des capacités d'exécution qui mettent en œuvre des méca-

nismes de déduction pour réaliser le passage de propositions générales à un cas particulier. A l'inverse de l'induction, la déduction est en effet un type de raisonnement qui « va » du général au particulier, de la cause aux effets ou encore du principe aux conséquences.

Cette étape permet ainsi de développer des capacité d'exécution en respectant rigoureusement des consignes imposées. Elle met ainsi en évidence des capacités « techniques » qui seront très utiles aux futurs ingénieurs dans leur mission d'encadrement d'équipes d'ouvriers et de techniciens.

1.4.3 Vérification du résultat

Etant donné un résultat obtenu par application d'une méthode générique pour résoudre un problème particulier, la troisième étape de la démarche MRV consiste à vérifier ce résultat par des méthodes alternatives ou complémentaires de celle déjà utilisée.

Pour un débutant, cette étape de vérification du résultat obtenu est assez difficile car il s'agit avant tout de remettre en cause son propre travail. Dans le meilleur des cas, le débutant estime que la vérification consiste simplement à refaire les mêmes calculs, le même raisonnement ou la même démarche : c'est effectivement la moindre des choses que de ré-appliquer la méthode pour vérifier qu'on ne s'est pas trompé en l'appliquant la première fois. Mais la vérification consiste plutôt à changer de point de vue sur le problème et à utiliser d'autres méthodes pour « estimer » la validité du résultat.

Il peut effectivement exister plusieurs méthodes alternatives pour résoudre un même problème. Résoudre alors le problème par deux méthodes différentes et obtenir le même résultat renforce bien entendu la « confiance » en ce résultat. Mais dans bien des cas, il s'agit plutôt de méthodes complémentaires, le plus souvent sous forme d'heuristiques, qui permettent de détecter que le résultat est certainement faux, comme par exemple la preuve par 9 en calcul élémentaire ou l'analyse dimensionnelle en physique. Et si de telles méthodes complémentaires ne détectent pas que le résultat est faux, alors ça renforce ici encore la « confiance » que l'on peut avoir dans le résultat obtenu.

Cette étape permet ainsi de développer l'esprit critique des futurs ingénieurs en insistant sur le souci de vérification systématique de ses propres résultats comme de ceux provenant d'autres sources (collègues, articles, internet...).

1.5 Mise en œuvre

La démarche MRV est mise en œuvre dans le cours d'« **Initiation à l'algorithmique** » du semestre S1 à l'ENIB (1.5.1). Lors des contrôles, elle conduit à une triple évaluation des exercices selon une notation adaptée (1.5.2) dans le cadre d'un contrôle continu systématique (1.5.3).

1.5.1 Contexte

Les « **Questionnements de cours** » qui accompagnent le cours d'« **Initiation à l'algorithmique** » du semestre S1 à l'ENIB sont conçus de façon plutôt « ascendante » (de l'exemple au concept) et se veulent complémentaires des notes de cours qui, elles, sont conçues plutôt classiquement de façon « descendante » (du concept à l'exemple).

Chaque questionnaire concerne un point particulier du cours ; il est structuré en 5 parties de la manière suivante :

1. Exemple : dans cette partie, des questions « simples » sont posées sur un problème « connu » de la « vie courante » afin d'introduire le concept informatique sous-jacent.

On y trouvera des exemples tels qu'aller au restaurant, ranger un meuble à tiroirs, analyser les sorties d'un circuit logique, compter avec BOBBY LAPOINTE en base « bibi », déterminer sa mention au bac, planter un clou, ranger des rondins de bois, cuisiner un quatre-quarts aux pépites de chocolat, jouer aux tours de Hanoï ou encore trier un jeu de cartes.

Cette partie est principalement traitée de manière informelle par les étudiants eux-mêmes, individuellement ou en groupe.

2. Généralisation : dans cette partie, les concepts informatiques sous-jacents sont présentés et introduits à l'aide de questions plus « informatiques ». On y aborde les concepts d'algorithmique, d'affectation, de calculs booléens, de codage des nombres, de tests et d'alternatives, de boucles, de spécification de fonction, de récursivité ou encore de manipulation de séquences.

En général, cette partie est traitée par l'enseignant.

3. Applications : des exemples « simples » d'application sont ensuite proposés. Le premier exemple est en général traité *in extenso* par l'enseignant en suivant la démarche MRV, les autres par les étudiants, en groupe ou individuellement.
4. Entraînement : cette partie est une préparation à l'évaluation qui a lieu en début de séance suivante. Les étudiants y travaillent chez eux entre les deux séances, individuellement ou en groupe.

- (a) Enoncé : on présente ici le problème que l'on souhaite traiter tel que le calcul en base « Shadok », le calcul de facteurs de conversion entre unités physiques, l'établissement de la table de vérité d'une expression logique, l'écriture d'un nombre réel selon la norme IEEE 754, la détermination de la valeur d'une fonction continue et linéaire par morceaux, le calcul d'un développement limité selon une certaine précision, le dessin d'un motif géométrique composé de polygones réguliers, la spécification d'une fonction connue (un « grand classique » de la programmation), le parcours d'un arbre binaire ou encore le tri d'un annuaire selon différents critères.

- (b) Exemple : un exemple est traité en détail dans cette partie en suivant la démarche MRV. Ce sont de tels exemples qui ont été présentés dans la section 1.2 du présent document.

- (c) Questions : 24 questions de même difficulté sont proposées ici pour permettre à chaque étudiant de s'entraîner sur le problème à traiter. La résolution de 2 ou 3 de ces exemples l'aide ainsi à induire (à faire émerger) la méthode générique qui est attendue ainsi qu'à mener explicitement les vérifications souhaitées.

Le jour de l'évaluation, chaque étudiant traite individuellement une des 24 questions tirée au sort le jour du contrôle (une question différente par élève). Lors de cette évaluation, tous documents, calculatrices, téléphones et ordinateurs sont interdits. A la fin de chaque contrôle, il est demandé à chaque étudiant de s'auto-évaluer pour chacune des étapes de la méthode MRV selon une grille de notation à 4 niveaux (voir section 1.5.2 suivante). Enfin, une correction est proposée par l'enseignant juste après le contrôle, « à chaud ».

5. Révisions : Cette partie fait le lien entre les questionnements de cours et les notes de cours.

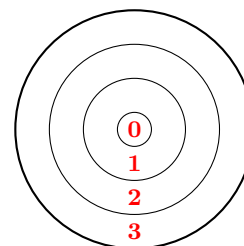
1.5.2 Triple évaluation

Chaque contrôle donne lieu à une triple évaluation de la part de l'enseignant : une évaluation concerne la qualité de l'explicitation de la méthode générique (M), une autre la qualité du résultat

obtenu (R) et la troisième la pertinence de la vérification du résultat (V). Ainsi, le résultat à la question posée, dont se contentent le plus souvent les étudiants, n'est plus le seul élément de réponse attendu : il est également demandé aux étudiants d'explicitier la méthode utilisée ainsi que les vérifications menées.

La grille de notation adoptée doit permettre de « soulager » l'enseignant dans sa tâche de correction et d'aider les étudiants à mener leurs propres évaluations. Un exercice cherchant à évaluer un objectif particulier, la notation exprime alors « simplement » la distance qui reste à parcourir pour atteindre cet objectif. Quatre « distances » sont ainsi pré-définies selon une métaphore de la cible :

- 0 : « en plein dans le mille ! » → l'objectif est atteint
- 1 : « pas mal ! » → on est proche de l'objectif
- 2 : « juste au bord de la cible ! » → on est encore loin de l'objectif
- 3 : « la cible n'est pas touchée ! » → l'objectif n'est pas atteint



Ayant choisi de ne garder qu'un petit nombre de niveaux pour « faciliter » l'évaluation, le choix de 4 niveaux a finalement été préféré à 2, 3 ou 5 niveaux :

- une notation sur 2 niveaux (*tout ou rien*) est un peu trop caricaturale ;
- avec un (petit) nombre impair de niveaux (3 ou 5), l'expérience montre que, dans le doute, le correcteur a tendance à choisir plus facilement le niveau du milieu (1 ou 3) alors qu'avec un nombre pair de niveau (ici 4), il doit « choisir son camp » : objectif plutôt atteint (0 ou 1) ou plutôt raté (2 ou 3).

En fait, il existe un cinquième niveau qui correspond à une absence au contrôle, sanctionnée par la note 4 (l'objectif n'a pas été visé), dite note minimale.

Ainsi, et pour changer de point de vue sur la notation, le contrôle est réussi lorsqu'on a 0 ! Il n'y a pas non plus de 1/2 point ou de 1/4 de point : le seul barème possible ne comporte que 4 niveaux : 0, 1, 2 et 3. On ne cherche donc pas à « grappiller » des points :

- on peut avoir 0 (objectif atteint) et avoir fait une ou deux erreurs bénignes en regard de l'objectif recherché ;
- on peut avoir 3 (objectif non atteint) et avoir quelques éléments de réponse corrects mais sans grand rapport avec l'objectif.

Pour obtenir une note plus « classique » (ie. une note sur 20 : n_{20}), il suffit de prendre le complément à 4 de la note sur 0 (n_0) et de le multiplier par 5 :

		n_0	n_{20}	signification
$n_{20} = (4 - n_0) \times 5$ soient les équivalences :		0	20	l'objectif est atteint
		1	15	on est proche de l'objectif
		2	10	on est encore loin de l'objectif
		3	5	l'objectif n'est pas atteint
		4	0	l'objectif n'a pas été visé

Dans ce contexte, avoir 20/20 ne signifie pas qu'on est génial ou que c'est parfait, cela signifie « juste » qu'on a atteint un objectif fixé, et c'est déjà beaucoup !

1.5.3 Contrôle continu

À l'ENIB, le cours d'informatique du semestre S1 est un enseignement de 42h réparties régulièrement sur 14 semaines : 1h30 de cours-td en salle banalisée toutes les semaines (par

groupes de 36 étudiants maximum) et 3h de laboratoire en salle informatique toutes les deux semaines (par groupes de 24 étudiants maximum). Chaque séance donne lieu a priori à une évaluation :

- 1 QCM de questions de cours une séance de cours-td sur deux, soient 7 QCM de 5' chacun, en fin de séance ;
- 1 contrôle MRV une séance de cours-td sur deux, soient 7 MRV de 30' chacun, en début de séance ;
- 1 contrôle sur machine à chaque séance de laboratoire, soient 7 LABO de 30' chacun, en début de séance.

Chaque exercice est évalué selon la grille de notation à 4 niveaux décrite à la section 1.5.2 précédente.

L'accumulation et la fréquence des contrôles, notés d'une séance à l'autre, permettent un suivi plus régulier et plus fin des apprentissages des étudiants. Ceux-ci travaillent plus et plus régulièrement en développant au fur et à mesure leurs capacités d'abstraction et leur esprit critique. Et de leur avis même, ils ont l'impression au bout du compte de mieux maîtriser leurs apprentissages.

1.6 Conclusion

La démarche MRV (Méthode–Résultat–Vérification) mise en place dans le cadre du cours d'« **Initiation à l'algorithmique** » du semestre S1 à l'ENIB, cherche à développer, à travers l'acquisition de compétences thématiques en informatique, des compétences plus transversales, nécessaires aux futurs ingénieurs : la capacité d'abstraction, la rigueur applicative et l'esprit critique.

C'est pourquoi la démarche MRV repose sur trois étapes bien distinctes :

1. l'explicitation d'une méthode générique de résolution d'une famille de problèmes équivalents pour développer la capacité d'abstraction de l'étudiant,
2. l'application de la méthode proposée à un cas particulier pour développer sa rigueur applicative et
3. la vérification du résultat ainsi obtenu pour développer son esprit critique.

Sa mise en œuvre à travers un contrôle continu systématique, quoique récente, permet d'entrevoir quelques évolutions encourageantes. Les étudiants ne se contentent plus d'un simple résultat répondant strictement à la question posée mais s'engagent avec plus d'intérêt dans la généralisation de leur méthode et dans la remise en cause de leur propre résultat. Ils travaillent plus et plus régulièrement et enfin, ils ont l'impression de mieux maîtriser leurs apprentissages.

Il reste que cette démarche est lourde à mettre en place pour un enseignant isolé et seules des équipes pédagogiques constituées pourront s'engager sereinement dans cette voie exigeante.

1.6. CONCLUSION

Chapitre 2

Premiers pas

2.1 Organisation du document

2.2 Calculette Python

Deuxième partie

Instructions de base

Chapitre 3

Affectation

3.1 Rappels de cours

3.1.1 Variables

En informatique, l'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses et pour accéder à ces données, il est pratique de les nommer plutôt que de connaître explicitement leur adresse en mémoire.

Une donnée apparaît ainsi sous un nom de variable : on dit que la variable dénote une valeur (fait référence à une valeur). Pour la machine, il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive où est stockée une valeur bien déterminée qui est la donnée proprement dite. En informatique, une variable possède à un moment donné une valeur et une seule.

Une variable peut ainsi être vue comme une case en mémoire vive, que le programme va repérer par une étiquette (une adresse ou un nom). Pour avoir accès au contenu de la case (la valeur de la variable), il suffit de la désigner par son étiquette : c'est-à-dire soit par son adresse en mémoire, soit par son nom.

Définition 1 (variable). *Une variable est un objet informatique qui associe un nom à une valeur qui peut éventuellement varier au cours du temps.*

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

- Un nom de variable est une séquence de lettres (a...z , A...Z) et de chiffres (0...9), qui doit toujours commencer par une lettre.
Exemples : `x`, `x1`, `x2`, `theta`, `tmp`, `mot`, `pression`, `longitude`
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits, à l'exception du caractère `_` (souligné).
Exemples : `vitesse_angulaire`, `une_variable`, `une_autre_variable`
- La « casse » est significative : les caractères majuscules et minuscules sont distingués. Ainsi, `python`, `Python`, `PYTHON` sont des variables différentes.
- Par convention, on écrira l'essentiel des noms de variable en caractères minuscules (y compris la première lettre). On n'utilisera les majuscules qu'à l'intérieur même du nom pour en augmenter éventuellement la lisibilité, comme dans `programmePython` ou `angleRotation`.

Une variable dont la valeur associée ne varie pas au cours du programme (on parle alors de constante) pourra être écrite entièrement en majuscule, par exemple `PI` ($\pi = 3.14$) ou `ROUGE` (la couleur rouge).

- Le langage lui-même peut se réserver quelques noms comme c'est le cas pour `PYTHON` (Table 3.1). Ces mots réservés ne peuvent donc pas être utilisés comme noms de variable.

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

TABLE 3.1 – Mots réservés en PYTHON 3

3.1.2 Attribuer une valeur

Une fois nommée, il est souvent nécessaire de modifier la valeur de la donnée référencée par une variable. C'est le rôle de l'instruction d'affectation.

Définition 2 (affectation). *L'affectation est l'opération qui consiste à attribuer une valeur à une variable.*

L'instruction d'affectation est notée `=` en PYTHON : `variable = valeur`. Le nom de la variable à modifier est placé dans le membre de gauche du signe `=`, la valeur qu'on veut lui attribuer dans le membre de droite. Le membre de droite de l'affectation est d'abord évalué sans être modifié puis la valeur obtenue est affectée à la variable dont le nom est donné dans le membre de gauche de l'affectation ; ainsi, cette opération ne modifie que le membre de gauche de l'affectation. L'affectation n'est donc pas une opération commutative (symétrique) : $(a = b) \neq (b = a)$. En effet, avec l'instruction `a = b`, on modifie la valeur de `a` et pas celle de `b` tandis qu'avec l'instruction `b = a`, on modifie `b` mais pas `a`.

Le membre de droite peut être une constante ou une expression évaluable.

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

<code>booléen = False</code>	<code>autreBooléen = True</code>
<code>entier = 3</code>	<code>autreEntier = -329</code>
<code>reel = 0.0</code>	<code>autreReel = -5.4687e-2</code>
<code>chaîne = "salut"</code>	<code>autreChaîne = 'bonjour, comment ça va ?'</code>
<code>tableau = [5,2,9,3]</code>	<code>autreTableau = ['a', [6,3.14], [x,y, [z,t]]]</code>
<code>matrice = [[1,2], [6,7]]</code>	<code>autreMatrice = [[1,2], [3,4], [5,6], [7,8]]</code>
<code>nUpLet = 4,5,6</code>	<code>autreNUpLet = "e", True, 6.7, 3, "z"</code>
<code>dictionnaire = {}</code>	<code>autreDictionnaire = {"a":7, "r":-8}</code>

En PYTHON, la valeur que l'on affecte à une variable impose dynamiquement le type de la variable (Table 3.2). Ainsi, si l'on affecte la valeur 3.14 à la variable `x` (`x = 3.14`),

Types de base en Python		
nom	type	exemples
booléen	bool	False, True
entier	int	3, -7
réel	float	3.14, 7.43e-3
chaîne	str	'salut', "l'eau"
n-uplet	tuple	1,2,3
liste	list	[1,2,3]
dictionnaire	dict	{'a':4, 'r':8}

TABLE 3.2 – Principaux types de base en PYTHON

celle-ci dénote alors un réel (type `float` en PYTHON); si par contre, on lui affecte la valeur `'salut'` (`x = 'salut'`), elle dénote alors une chaîne de caractères (type `str` en PYTHON).

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

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

L'expression du membre de droite peut faire intervenir la variable du membre de gauche comme dans `i = i+1`. Dans cet exemple, on évalue d'abord le membre de droite (`i+1`) puis on attribue la valeur obtenue au membre de gauche (`i`); ainsi, à la fin de cette affectation, la valeur de `i` a été augmentée de 1 : on dit que `i` a été incrémenté de 1 et on parle d'incrément de la variable `i`. Le langage PYTHON propose un opérateur d'incrément (`+=`) et d'autres opérateurs d'affectation qui peuvent toujours se ramener à l'utilisation de l'opérateur `=`, l'opérateur d'affectation de base (Table 3.3).

Affectations en Python		
<code>a = b</code>		
<code>a += b</code>	<code>≡</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>≡</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>≡</code>	<code>a = a * b</code>
<code>a /= b</code>	<code>≡</code>	<code>a = a / b</code>
<code>a %= b</code>	<code>≡</code>	<code>a = a % b</code>
<code>a **= b</code>	<code>≡</code>	<code>a = a ** b</code>

TABLE 3.3 – Affectations en PYTHON

Avec l'exemple de l'incrément (`i = i+1`), on constate que l'affectation est une opération typiquement informatique qui se distingue de l'égalité mathématique. En effet, en mathématique une expression du type `i = i+1` se réduit en `0 = 1`! Alors qu'en informatique, l'expression `i = i+1` conduit à ajouter 1 à la valeur de `i` (évaluation de l'expression `i+1`), puis à donner cette nouvelle valeur à `i` (affectation). L'affectation peut ainsi être vue comme un opérateur temporel : il y a un avant et un après l'affectation.

L'affectation a donc pour effet de réaliser plusieurs opérations en mémoire :

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

3.1.3 Séquences d'affectations

Un apprenti informaticien a qui on demandait d'échanger (*swap*) les valeurs de 2 variables x et y proposa la suite d'instructions suivante :

$x = y$ et eut la désagréable surprise de constater que les valeurs des variables
 $y = x$ n'étaient pas permutées après cette séquence d'affectations.

En effet, pour fixer les idées supposons qu'initialement $x = 10$ et $y = 20$. L'affectation $x = y$ conduit à évaluer y puis à attribuer la valeur de y (20) à x : x vaut maintenant 20. La deuxième affectation ($y = x$) commence par évaluer x puis à attribuer la valeur de x (20) à y . Après ces 2 affectations, x et y sont donc identiques et non permutées ! Pour effectuer la permutation, l'apprenti informaticien aurait pu utiliser une variable temporaire (que nous nommerons `tmp`) et exécuter la séquence d'instructions suivante :

	La première affectation (<code>tmp = x</code>) permet de stocker la valeur initiale de x
<code>tmp = x</code>	(10), la deuxième (<code>x = y</code>) attribue à x la valeur de y (20) et la troisième (<code>y</code>
<code>x = y</code>	<code>= tmp</code>) attribue à y la valeur de <code>tmp</code> , c'est-à-dire la valeur initiale de x (10).
<code>y = tmp</code>	Ainsi, les valeurs finales de x et y (20 et 10) sont bien permutées par rapport
	aux valeurs initiales (10 et 20).

En PYTHON, les *n*-uplets permettent d'écrire plus simplement la permutation de deux variables :
`x, y = y, x`.

L'exemple précédent de la permutation illustre la possibilité de réaliser des calculs plus ou moins compliqués à l'aide d'une séquence d'affectations bien choisies. Mais ce sont les tests (Chapitre 4) et les boucles (Chapitre 5) qui nous permettront d'aborder des algorithmes réutilisables et plus robustes, en améliorant l'expressivité du programmeur.

Les sections suivantes de ce chapitre mettent en œuvre l'instruction d'affectation dans différents domaines thématiques.

3.2 Vie courante : prix d'un livre

3.2.1 Objectifs

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : déterminer le prix d'un livre compte-tenu de réductions et des taxes.

3.2.2 Syntaxe Python

`variable = expression`

3.2.3 Enoncé

Un libraire propose une réduction de 3.5% sur le prix hors taxes (HT) d'un livre à 12.35 € HT. Sachant que la taxe sur la valeur ajoutée (TVA) sur les livres est de 5.5%, proposer une

instruction de type « affectation » qui permettra de calculer le prix final toutes taxes comprises (TTC) pour le client.

3.2.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier numériques, pour ne considérer que les variables associées. Ainsi, on ne s'intéressera pas à la valeur « 12.35 € » mais à ce qu'elle qualifie : c'est-à-dire le prix hors taxes du livre, le « livre » étant lui-même un cas particulier de produit. Et ainsi de suite pour chaque donnée spécifique du problème posé à laquelle on attribuera un nom.

Il s'agit donc ici de calculer le prix TTC (noté `ttc`) d'un produit connaissant son prix HT (noté `ht`), la TVA (notée `tva` et exprimée sous la forme d'un pourcentage) sur ce type de produit et la réduction éventuelle (notée `r` et exprimée sous la forme d'un pourcentage du prix HT) proposée par le vendeur sur ce produit.

Sachant qu'une réduction se soustrait du prix HT initial et qu'une taxe s'ajoute au prix hors taxes, on a ainsi : $ttc = ht * (1 - r/100) * (1 + tva/100)$.

3.2.5 Résultat

Appliquer la méthode précédente revient simplement ici à initialiser les variables `ht`, `r` et `tva` aux données du problème particulier.

On applique donc la méthode précédente au livre à 12.35 € HT (`ht = 12.35`) avec une réduction de 3.5% (`r = 3.5`) et une TVA à 5.5% (`tva = 5.5`).

Listing 3.1 – prix d'un livre

Compte-tenu de ces valeurs, le code PYTHON ci-contre permet de calculer le prix final TTC demandé (12.57 €).

```
1 ht, r, tva = 12.35, 3.5, 5.5
2 ttc = ht*(1-r/100)*(1+tva/100)

>>> ttc
12.57322625
```

Remarque : on n'a pas cherché à effectuer « à la main » les calculs numériques : PYTHON les fera mieux que nous ; et surtout, on n'a pas cherché non plus à particulariser la 2^{ème} ligne du code en `ttc = 12.35*0.965*1.055` : la forme plus abstraite `ttc = ht*(1-r/100)*(1+tva/100)` restera identique pour tout autre type de produit, de réduction et de TVA.

3.2.6 Vérification

Une vérification possible est liée à l'ordre de grandeur du résultat obtenu.

Compte-tenu des taux respectifs de la réduction et de la TVA ($r < tva$), le prix final sera supérieur au prix HT et inférieur au prix TTC sans réduction. On vérifie bien cette propriété : $12.35 < 12.57 < 12.35 \cdot (1 + 0.055) = 13.03$, le prix obtenu ici (12.57 €) a donc de fortes chances d'être correct.

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de produits, de taux de TVA ou encore de réductions. C'est l'objet de la section 3.2.7 suivante.

3.2.7 Généricité

Pour tester la généricité de la méthode précédente, calculer à l'aide du code PYTHON précédent, les prix TTC dans les cas suivants.

1. Un cinéma propose des places de cinéma, taxées à 7%, avec une réduction de 44.75% sur le prix HT de 9.30€. Quel est le prix de la place de cinéma pour le client ?
2. Une grande surface vend 3 tablettes de chocolat de 100g chacune pour le prix de 2 sachant que le chocolat, soumis à une TVA de 19.6%, est à 9.90€/kg. Quel est le prix des 3 tablettes de chocolat pour le client ?

3.2.8 Entraînement

Dans le même esprit, utiliser une instruction d'affectation pour déterminer :

1. la durée totale du trajet BREST-RENNES, en car de BREST (10h00) à QUIMPER (11h20) puis en TGV de QUIMPER (11h40) à RENNES (13h55),
2. l'augmentation de la facture de fuel d'un particulier pour une cuve de 1400 l si le fuel domestique, anciennement à 0.938 €/l, augmente de 5%.

3.3 Jeux : lancer de dés

3.3.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : simuler un lancer de dés.

3.3.2 Syntaxe Python

variable = expression

range(start,stop,step) retourne une liste d'entiers compris entre **start** inclus (= 0 par défaut) et **stop** exclu, par pas de **step** (= 1 par défaut).

```
>>> range(0,4,1)          >>> range(3)
[0, 1, 2, 3]              [0, 1, 2]
>>> range(0,4)            >>> range(3,9,2)
[0, 1, 2, 3]              [3, 5, 7]
>>> range(4)              >>> range(7,0,-1)
[0, 1, 2, 3]              [7, 6, 5, 4, 3, 2, 1]
```

randrange(start,stop,step) retourne un élément choisi aléatoirement dans la liste retournée par la fonction **range(start,stop,step)**. Il est nécessaire d'importer la fonction **randrange** depuis le module **random** : **from random import randrange**.

```
>>> from random import randrange          >>> from random import randrange
>>> randrange(0,4,1)                      >>> randrange(3)
1                                          2
>>> randrange(0,4)                        >>> randrange(3,9,2)
0                                          5
>>> randrange(4)                          >>> randrange(7,0,-1)
2                                          3
```

3.3.3 Énoncé

Proposer une instruction de type « affectation » qui permettra de simuler un lancer de dé à 6 faces. On pourra utiliser la fonction `randrange` de la section 3.3.2 précédente.

3.3.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier numériques, pour ne considérer que les variables associées. Ainsi, on généralisera le « dé à six faces » en un tirage aléatoire d'un entier compris entre deux autres entiers à chacun desquels on attribuera un nom.

Il s'agit donc ici de déterminer un entier `de` compris entre 2 nombres `min` et `max` inclus, par pas de `step`. On peut donc directement utiliser la fonction `randrange` :
`de = randrange(min,max+1,step)`.

3.3.5 Résultat

Appliquer la méthode précédente revient simplement ici à initialiser les variables `min`, `max` et `step` aux données du problème particulier.

On applique donc la méthode précédente au dé à 6 faces : `min = 1`, `max = 6` et `step = 1`.

Listing 3.2 – lancer de dé

Compte-tenu de ces valeurs, le code PYTHON ci-contre permet de simuler un lancer de dé à 6 faces.

```
1 from random import randrange
2 min, max, step = 1, 6, 1
3 de = randrange(min,max+1,step)

>>> de
4
```

Remarque : on n'a pas cherché à particulariser et simplifier la 3^{ème} ligne du code en `de = randrange(1,7)` : la forme plus abstraite `de = randrange(min,max+1,step)` restera identique pour tout autre type de « dé ».

3.3.6 Vérification

La fonction `randrange` est une fonction d'un module standard de PYTHON (le module `random`) : il y a peu de doutes sur le fait qu'elle fonctionne correctement. On vérifiera simplement ici que le code proposé ne « sort » pas des valeurs inférieures à 1 ou supérieures à 6 en le testant plusieurs fois de suite ou, au contraire, « sort » bien de temps en temps les valeurs

extrêmes 1 et 6.

```
>>> from random import randrange
>>> min, max, step = 1, 6, 1
>>> de = randrange(min,max+1,step)
>>> de
2
>>> de = randrange(min,max+1,step)
>>> de
6
>>> de = randrange(min,max+1,step)
>>> de
3
>>> de = randrange(min,max+1,step)
>>> de
3

>>> from random import randrange
>>> min, max, step = 1, 6, 1
>>> de = randrange(min,max+1,step)
>>> de
1
>>> de = randrange(min,max+1,step)
>>> de
3
>>> de = randrange(min,max+1,step)
>>> de
5
>>> de = randrange(min,max+1,step)
>>> de
2
```

Une autre vérification nécessaire concerne la généricité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de « dés ». C'est l'objet de la section 3.3.7 suivante.

3.3.7 Généricité

Pour tester la généricité de la méthode précédente, simuler, à l'aide du code PYTHON précédent, le tirage aléatoire :

1. d'une carte d'un jeu de 32 cartes,
2. d'une boule du LOTO.

3.3.8 Entraînement

Dans le même esprit, utiliser l'affectation pour :

1. simuler le tirage aléatoire d'une carte d'une couleur donnée (♠, ♥, ♦ ou ♣) issue d'un jeu de 32 cartes, la couleur étant elle-même tirée au sort ;
2. calculer la somme de deux dés tirés chacun aléatoirement.

3.4 Textes : longueurs de chaînes de caractères

3.4.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : comparer les longueurs de deux chaînes de caractères.

3.4.2 Syntaxe Python

variable = expression

len(s) retourne le nombre d'éléments (la longueur, *length*) de la séquence **s**.

x in s teste si **x** est un élément de (appartient à, est dans) la séquence **s**.

s[i] retourne l'élément de rang **i** dans la séquence **s**.

Attention ! Par convention, les indices des éléments dans une séquence commencent à 0. Le premier élément a pour indice 0 (**s[0]**), le deuxième l'indice 1 (**s[1]**), le troisième l'indice 2 (**s[2]**) et ainsi de suite jusqu'au dernier qui a l'indice **n-1** (**s[n-1]**) si **n** est le nombre d'éléments dans la séquence (**n == len(s)**).

`s[i : j]` retourne la séquence comprise entre les rangs `i` inclus et `j` exclu de la séquence `s`.

`s1 + s2` retourne la séquence résultat de la concaténation (la mise bout à bout) des deux séquences `s1` et `s2`.

n-uplets :

```
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> len(s)
4
>>> 3 in s
False
>>> s[1]
7
>>> s[1:3]
(7, 2)
>>> s + (5,3)
(1, 7, 2, 4, 5, 3)
```

chaînes de caractères :

```
>>> s = '1724'
>>> type(s)
<type 'str'>
>>> len(s)
4
>>> '3' in s
False
>>> s[1]
'7'
>>> s[1:3]
'72'
>>> s + '53'
'172453'
```

listes :

```
>>> s = [1,7,2,4]
>>> type(s)
<type 'list'>
>>> len(s)
4
>>> 3 in s
False
>>> s[1]
7
>>> s[1:3]
[7, 2]
>>> s + [5,3]
[1, 7, 2, 4, 5, 3]
```

3.4.3 Enoncé

Proposer une instruction de type « affectation » qui permettra de tester si les deux chaînes de caractères `'bonjour'` et `'salut'` ont le même nombre d'éléments. On pourra utiliser la fonction `len` de la section 3.4.2 précédente.

3.4.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier alphanumériques, pour ne considérer que les variables associées. Ainsi, on ne s'intéressera pas aux valeurs `'bonjour'` et `'salut'` mais plutôt à ce qu'elles qualifient : deux chaînes de caractères auxquelles on attribuera un nom chacune.

Il s'agit donc ici de comparer les longueurs de deux chaînes `s1` et `s2`. La longueur d'une chaîne de caractères s'obtient à l'aide de la fonction `len` : `len(s1)` ou `len(s2)`. On affecte donc simplement la comparaison (`len(s1) == len(s2)`) à la réponse souhaitée :

`compar = (len(s1) == len(s2))`; `compar` prendra donc la valeur booléenne `True` ou `False` selon que les longueurs sont égales ou non.

3.4.5 Résultat

Appliquer la méthode précédente revient simplement ici à initialiser les variables `s1` et `s2` aux données du problème particulier.

On applique donc la méthode précédente avec les chaînes : `s1 = 'bonjour'` et `s2 = 'salut'`.

Listing 3.3 – comparaison de longueurs

Compte-tenu de ces valeurs, le code PYTHON ci-contre permet de comparer les longueurs de ces deux chaînes.

```
1 s1, s2 = 'bonjour', 'salut'
2 compar = (len(s1) == len(s2))

>>> compar
False
```

Remarque : on n'a pas cherché à particulariser la 2^{ème} ligne du code en `compar = (len('bonjour') == len('salut'))` : la forme plus abstraite `compar = (len(s1) == len(s2))` restera identique pour tout autre couple de chaînes.

3.4.6 Vérification

La vérification est ici immédiate puisqu'on peut calculer « à la main » la longueur de chacune de ces chaînes : il y a 7 caractères dans la chaîne 'bonjour' et 5 dans la chaîne 'salut'. Elles ne sont donc pas de la même longueur et le résultat de la comparaison vaut `False` : ce que donne bien l'exécution du code précédent.

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de chaînes de caractères. C'est l'objet de la section 3.4.7 suivante.

3.4.7 Généralité

Pour tester la généralité de la méthode précédente, déterminer, à l'aide du code PYTHON précédent, si les chaînes suivantes sont de même longueur ou non :

1. la chaîne 'salut monde' et la chaîne 'hello' concaténée avec la chaîne 'world',
2. la chaîne 'démocratie' et la chaîne 'anticonstitutionnellement' considérée uniquement entre les indices 4 et 8 inclus.

3.4.8 Entraînement

Dans le même esprit, utiliser l'affectation pour :

1. tester si la chaîne 'bonjour' a moins de caractères que la chaîne 'au revoir' ;
2. tester si la chaîne 'bonjour' est plus petite que la chaîne 'au revoir' selon l'ordre lexicographique.

3.5 Nombres : quotient et reste

3.5.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : calculer le quotient et le reste de la division entière de deux entiers.

3.5.2 Syntaxe Python

`variable = expression`

`a//b` retourne le quotient `q` de la division entière de `a` par `b` : `a == b*q + r`.

`a%b` retourne le reste `r` de la division entière de `a` par `b` : `a == b*q + r`.

3.5.3 Énoncé

Proposer une instruction de type « affectation » qui permettra de calculer le quotient et le reste de la division entière $135 \div 2$. On pourra utiliser les opérateurs `//` et `%` de la section 3.5.2 précédente.

3.5.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier numériques, pour ne considérer que les variables associées. Ainsi, on ne

s'intéressera pas aux valeurs 135 et 2 mais plutôt à ce qu'elles qualifient : deux entiers auxquels on attribuera un nom chacun.

Il s'agit donc ici de déterminer le quotient q et le reste r de la division entière de deux nombres a et b : $a \div b$. On peut utiliser simplement les opérateurs prédéfinis `//` et `%` :

`q, r = a//b, a%b`.

3.5.5 Résultat

Appliquer la méthode précédente revient simplement ici à initialiser les variables `a` et `b` aux données du problème particulier.

On applique donc la méthode précédente avec les entiers `a = 135` et `b = 2`.

Compte-tenu de ces valeurs, le code PYTHON ci-contre permet de déterminer le quotient `q` et le reste `r` recherchés.

Listing 3.4 – quotient et reste

```
1 a, b = 135, 2
2 q, r = a//b, a%b

>>> q, r
(67, 1)
```

Remarque : on n'a pas cherché à particulariser la 2^{ème} ligne du code en `q,r = 135//2, 135%2`; la forme plus abstraite `q, r = a//b, a%b` restera identique pour tout autre division entière.

3.5.6 Vérification

La vérification est ici immédiate puisqu'on peut calculer « de tête » le quotient (67) et le reste (1) de la division entière de 135 par 2.

Une vérification plus générale consiste à vérifier la relation `a == b*q + r` après avoir calculé `q` et `r`.

```
1 a, b = 135, 2
2 q, r = a//b, a%b

>>> a == b*q + r
True
```

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de divisions entières. C'est l'objet de la section 3.5.7 suivante.

3.5.7 Généralité

Pour tester la généralité de la méthode précédente, déterminer, à l'aide du code PYTHON précédent, les quotients et restes des divisions suivantes :

1. $3553 \div 11$
2. $375375 \div 11$

3.5.8 Entraînement

Dans le même esprit, utiliser l'affectation pour déterminer :

1. le nombre dont le quotient et le reste sont respectivement 13 et 7 lors d'une division par 17;
2. le diviseur de 345 si le quotient vaut 12 et le reste 9.

3.6 Figures : distance entre deux points

3.6.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : calculer la distance entre deux points du plan.

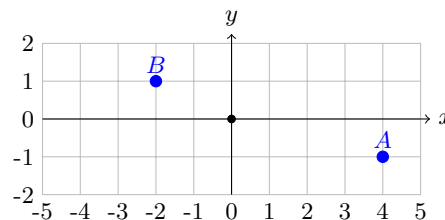
3.6.2 Syntaxe Python

`variable = expression`

`sqrt(x)` (*square root*) retourne la racine carrée de x (\sqrt{x}). Il est nécessaire d'importer la fonction `sqrt` depuis le module `math` : `from math import sqrt`.

3.6.3 Enoncé

Proposer une instruction de type « affectation » qui permettra de calculer la distance entre les deux points A et B du plan ci-dessous.



3.6.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier numériques, pour ne considérer que les variables associées. Ainsi, on ne s'intéressera pas aux valeurs des coordonnées $(4, -1)$ ou $(-2, 1)$ mais plutôt à ce qu'elles qualifient : deux points du plan auxquels on attribuera un nom chacun.

Il s'agit donc ici de déterminer la distance d entre deux points P_1 et P_2 de \mathbb{R}^2 respectivement de coordonnées (x_1, y_1) et (x_2, y_2) , à savoir $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, d'où l'affectation : `d = sqrt((x2-x1)**2 + (y2-y1)**2)`.

3.6.5 Résultat

Appliquer la méthode précédente revient simplement ici à initialiser les variables `x1`, `y1`, `x2` et `y2` aux données du problème particulier.

On applique donc la méthode précédente avec les coordonnées des points A et B : `(x1, y1) = (4, -1)` et `(x2, y2) = (-2, 1)`.

Compte-tenu de ces valeurs, le code PYTHON ci-contre permet de calculer la distance d entre les points A et B ($AB \approx 6.32$).

Listing 3.5 – distance entre 2 points

```
1 from math import sqrt
2 x1, y1 = 4, -1
3 x2, y2 = -2, 1
4 d = sqrt((x2-x1)**2 + (y2-y1)**2)

>>> d
6.324555320336759
```

Remarque : on n'a pas cherché à particulariser la 4^{ème} ligne du code en « préparant » les calculs « à la main » : $d = \text{sqrt}(40)$; la forme plus abstraite $d = \text{sqrt}((x2-x1)**2 + (y2-y1)**2)$ restera identique pour tout autre calcul de distance entre 2 points du plan.

3.6.6 Vérification

La vérification est ici immédiate puisqu'on peut vérifier rapidement que $d = \sqrt{40} \approx 6.32$.

Une autre vérification possible consiste à utiliser le module `turtle` (la tortue LOGO) de PYTHON et sa fonction `distance` pour déterminer cette distance et la comparer au calcul précédent.

La différence `delta` étant nulle, on obtient bien les mêmes résultats.

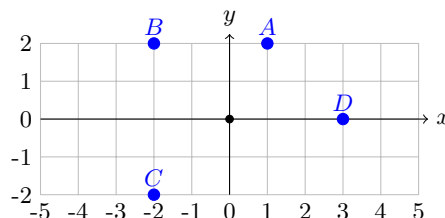
```
1 from turtle import *
2 up()
3 goto(x1,y1)
4 down()
5 delta = d - distance(x2,y2)

>>> delta
0.0
```

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de calculs de distances. C'est l'objet de la section 3.6.7 suivante.

3.6.7 Généralité

Pour tester la généralité de la méthode précédente, déterminer, à l'aide du code PYTHON précédent, les distances entre les points du diagramme ci-dessous : AB , AC , AD , BC , BD et CD .



3.6.8 Entraînement

Dans le même esprit, utiliser l'affectation pour :

1. vérifier que le triangle ABC formé par les points A , B et C de la figure ci-dessus (voir section 3.6.7 précédente) est un triangle rectangle ;
2. déterminer les coordonnées d'un point B sachant qu'il est situé à 4 unités de distance du point A de coordonnées $(1, -2)$ et que l'angle entre l'axe des x et \vec{AB} est de 60° .

3.7 Mathématiques : produit scalaire

3.7.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

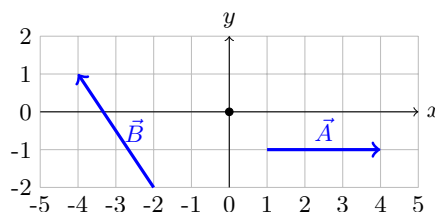
Secondaire : calculer le produit scalaire de deux vecteurs.

3.7.2 Syntaxe Python

`variable = expression`

3.7.3 Énoncé

On considère les 2 vecteurs \vec{A} et \vec{B} définis dans \mathbb{R}^2 comme le montre la figure ci-dessous.



Proposer une instruction de type « affectation » qui permettra de calculer le produit scalaire $\vec{A} \cdot \vec{B}$ de ces deux vecteurs.

3.7.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier numériques, pour ne considérer que les variables associées. Ainsi, on ne s'intéressera pas aux composantes numériques (3, 0) et (-2, 3) mais plutôt à ce qu'elles qualifient : deux vecteurs du plan auxquels on attribuera un nom chacun.

Il s'agit donc ici de déterminer le produit scalaire $\vec{V}_1 \cdot \vec{V}_2$ de 2 vecteurs \vec{V}_1 et \vec{V}_2 de \mathbb{R}^2 ayant pour composantes respectives (x_1, y_1) et (x_2, y_2) . Par définition, ce produit scalaire est un réel p qui a pour expression $p = x_1 \cdot x_2 + y_1 \cdot y_2$.

Connaissant les composantes respectivement (x_1, y_1) et (x_2, y_2) des vecteurs \vec{V}_1 et \vec{V}_2 , on détermine le produit scalaire p par une affectation simple : `p = x1*x2 + y1*y2`.

3.7.5 Résultat

On applique la méthode précédente aux vecteurs $\vec{V}_1 = \vec{A}$ et $\vec{V}_2 = \vec{B}$. Leurs composantes respectives se lisent directement sur la figure : $x_1 = (4) - (1) = 3$, $y_1 = (-1) - (-1) = 0$, $x_2 = (-4) - (-2) = -2$ et $y_2 = (1) - (-2) = 3$.

Compte-tenu de ces valeurs, le code ci-contre permet de calculer le produit scalaire $\vec{A} \cdot \vec{B}$ demandé.

Listing 3.6 – produit scalaire

```

1 x1, y1 = (4)-(1), (-1)-(-1)
2 x2, y2 = (-4)-(-2), (1)-(-2)
3 p = x1*x2 + y1*y2

```

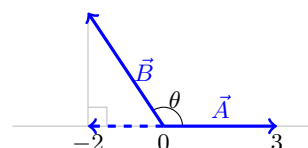
Remarque : on n'a pas cherché à effectuer « à la main » les calculs numériques : PYTHON les fera mieux que nous ; et surtout, on n'a pas cherché non plus à particulariser la 3^{ème} ligne

du code en $p = 3*(-2) + 0*3$: la forme plus abstraite $p = x1*x2 + y1*y2$ restera identique pour deux autres vecteurs quelconques de \mathbb{R}^2 .

3.7.6 Vérification

Une autre manière de déterminer le produit scalaire est donnée par la formule « géométrique » : $\vec{A} \cdot \vec{B} = |\vec{A}| \cdot |\vec{B}| \cdot \cos(\theta)$ où $|\vec{A}|$ et $|\vec{B}|$ représentent les modules des vecteurs \vec{A} et \vec{B} et θ l'angle formé par ces 2 vecteurs.

$|\vec{B}| \cdot \cos(\theta)$ représente donc la projection de \vec{B} sur \vec{A} . Or cette projection se lit très facilement sur la figure puisque \vec{A} est horizontal (parallèle à l'axe des abscisses avec $|\vec{A}| = 3$) : elle a pour valeur $(-4) - (-2) = -2$, d'où le produit scalaire $p = 3 \cdot (-2) = -6$.

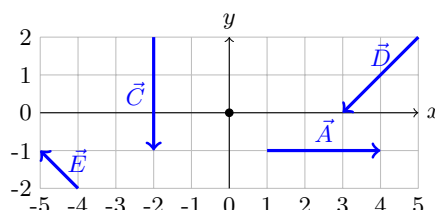


On obtient bien par le calcul analytique le résultat obtenu par la projection géométrique.

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de produits scalaires. C'est l'objet de la section 3.7.7 suivante.

3.7.7 Généralité

Pour tester la généralité de la méthode précédente, calculer, à l'aide du code PYTHON précédent, les produits scalaires suivants : $\vec{A} \cdot \vec{A}$, $\vec{A} \cdot \vec{C}$, $\vec{A} \cdot \vec{D}$, $\vec{A} \cdot \vec{E}$, $\vec{C} \cdot \vec{C}$, $\vec{C} \cdot \vec{D}$, $\vec{C} \cdot \vec{E}$, $\vec{D} \cdot \vec{D}$, $\vec{D} \cdot \vec{E}$ et $\vec{E} \cdot \vec{E}$. Les vecteurs \vec{A} , \vec{C} , \vec{D} et \vec{E} sont définis sur la figure ci-dessous.



Vérifier en particulier qu'on obtient bien les résultats attendus pour le produit scalaire d'un vecteur par lui-même et pour le produit scalaire de deux vecteurs perpendiculaires.

3.7.8 Entraînement

Dans le même esprit, utiliser l'affectation pour calculer :

- le produit vectoriel $\vec{A} \times \vec{B}$ de 2 vecteurs \vec{A} et \vec{B} de \mathbb{R}^3 de composantes respectives (a_1, a_2, a_3) et (b_1, b_2, b_3) ;
- le produit mixte $\vec{A} \cdot (\vec{B} \times \vec{C})$ de 3 vecteurs \vec{A} , \vec{B} et \vec{C} de \mathbb{R}^3 de composantes respectives (a_1, a_2, a_3) , (b_1, b_2, b_3) et (c_1, c_2, c_3) .

3.8 Physique : conversion d'unités

3.8.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : effectuer une conversion d'unités physiques.

3.8.2 Syntaxe Python

`variable = expression`

3.8.3 Énoncé

On veut convertir 3 nœuds marins (miles nautiques par heure) en kilomètres par heure (km/h).

Proposer une instruction de type « affectation » qui permettra de réaliser cette conversion.

3.8.4 Méthode

Pour expliciter la méthode qui nous permettra de résoudre un ensemble de problèmes équivalents et en particulier le problème posé dans l'énoncé, on commence par s'abstraire des données spécifiques, en particulier numériques, pour ne considérer que les variables associées. Ainsi, on ne s'intéressera pas au nombre de nœuds (3) à convertir, ni d'ailleurs aux nœuds eux-mêmes ou encore aux kilomètres par heure mais plutôt à ce qu'ils qualifient : deux unités physiques compatibles et une certaine quantité de l'une d'entre elles.

On cherche donc ici à convertir $n_1 \cdot u_1$ en $n_2 \cdot u_2$ où u_1 et u_2 sont des unités physiques compatibles qui dérivent de la même unité de base u_b du **Système international d'unités**.

$$\begin{cases} u_1 = a_1 \cdot u_b \\ u_2 = a_2 \cdot u_b \end{cases} \Rightarrow \begin{cases} n_1 \cdot u_1 = n_1 \cdot (a_1 \cdot u_b) = (n_1 \cdot a_1) \cdot u_b \\ n_2 \cdot u_2 = n_2 \cdot (a_2 \cdot u_b) = (n_2 \cdot a_2) \cdot u_b \end{cases} \Rightarrow \frac{n_1 \cdot u_1}{n_2 \cdot u_2} = \frac{n_1 \cdot a_1}{n_2 \cdot a_2}$$

Comme on cherche n_2 tel que $n_1 \cdot u_1 = n_2 \cdot u_2$, on a donc :

$$\frac{n_1 \cdot u_1}{n_2 \cdot u_2} = \frac{n_1 \cdot a_1}{n_2 \cdot a_2} = 1 \Rightarrow n_2 = n_1 \cdot \frac{a_1}{a_2}$$

où les coefficients a_i sont documentés dans le Système international d'unités par le **Bureau international des poids et mesures**.

Une fois connus les coefficients a_i , on détermine la quantité n_2 de l'unité u_2 par une affectation simple : `n2 = n1*a1/a2`.

3.8.5 Résultat

On applique la méthode précédente à la conversion proposée dans l'énoncé où u_1 représente les nœuds (miles nautiques par heure), u_2 les kilomètres par heure (km/h) et u_b les mètres par seconde (m/s). Le Système international d'unités fournit par ailleurs les facteurs de conversion a_1 (nd \rightarrow m/s) et a_2 (km/h \rightarrow m/s) : $a_1 = 1852/3600$ et $a_2 = 1000/3600$.

Compte-tenu de ces valeurs, le code ci-contre permet de calculer le nombre n_2 de kilomètres par heure en fonction du nombre n_1 de nœuds.

Listing 3.7 – **conversion d'unités**

```
1 a1 = 1852/3600
2 a2 = 1000/3600
3 n2 = n1*a1/a2
```

Remarque : on n'a pas cherché à effectuer « à la main » les calculs numériques : PYTHON les fera mieux que nous ; et surtout, on n'a pas cherché non plus à particulariser la 3^{ème} ligne du code en `n2 = n1*1852/1000` : la forme plus abstraite `n2 = n1*a1/a2` restera identique pour convertir des parsecs en années-lumière (longueurs), des gallons en barils (volumes) ou encore des électron-volts en frigories (énergies), seules les valeurs des coefficients a_i changeront (lignes 1 et 2 du code).

3.8.6 Vérification

Pour vérifier le résultat précédent, on peut comparer les valeurs obtenues par le calcul avec celles de quelques valeurs caractéristiques facilement évaluables « à la main » (exemples : $n_1 = 1 \text{ nd} \Rightarrow n_2 = 1.852 \text{ km/h}$ ou $n_1 = 1/1852 \text{ nd} \Rightarrow n_2 = 1/1000 \text{ km/h}$).

```
>>> n1 = 1
>>> a1, a2 = 1852/3600, 1000/3600
>>> n2 = n1*a1/a2
>>> n2
1.852

>>> n1 = 1/1852
>>> a1, a2 = 1852/3600, 1000/3600
>>> n2 = n1*a1/a2
>>> n2
0.0010000000000000002
```

On obtient bien par le calcul les résultats escomptés.

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de conversion d'unités. C'est l'objet de la section 3.8.7 suivante.

3.8.7 Généralité

Pour tester la généralité de la méthode précédente, effectuer, à l'aide du code PYTHON précédent, les conversions suivantes en s'appuyant sur le **Système international d'unités** :

1. des parsecs en années-lumière,
2. des gallons en barils,
3. des électron-volts en frigories.

3.8.8 Entraînement

Dans le même esprit, et en s'appuyant à nouveau sur le **Système international d'unités**, utiliser l'affectation pour convertir :

1. une température FAHRENHEIT ($^{\circ}\text{F}$) en une température CELSIUS ($^{\circ}\text{C}$).
2. une température RANKINE ($^{\circ}\text{Ra}$) en une température CELSIUS ($^{\circ}\text{C}$).

3.9 Informatique : préfixes binaires

3.9.1 Objectif

Principal : mettre en œuvre l'instruction d'affectation.

Secondaire : manipuler les puissances décimales et binaires de l'unité d'information.

3.9.2 Syntaxe Python

variable = **expression**

xn** calcule x^n .

3.9.3 Enoncé

Un bit est un chiffre binaire (*binary digit* : 0 ou 1). C'est l'unité élémentaire d'information. Un octet (*byte*) est une unité d'information composé de 8 bits. Proposer une instruction de type « affectation » qui permettra de déterminer le nombre d'octets dans 1 Go (1 giga-octet).

3.9.4 Méthode

En informatique, on parle souvent de kilo-octets (ko), méga-octets (Mo), giga-octets (Go), téra-octets (To) et autres péta-octets (Po). L'idée est donc ici est de savoir comment calculer ces différentes unités en fonction de l'octet.

Les préfixes kilo, méga, giga, téra, péta... sont définis par le **Système international d'unités** comme des puissances de 10^3 . En 1998, la **Commission Electrotechnique Internationale** institua les préfixes binaires plus adaptés à l'Informatique. Ce sont en fait des puissances de 2^{10} appelées kibi-octet, mébi-octet, gibi-octet, tébi-octet, pébi-octet... (Table 3.4).

		préfixes multiplicateurs							
préfixes décimaux		kilo	méga	giga	téra	péta	exa	zetta	yotta
symboles		ko	Mo	Go	To	Po	Eo	Zo	Yo
n		1	2	3	4	5	6	7	8
décimal	$(10^3)^n$	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
binaire	$(2^{10})^n$	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}	2^{70}	2^{80}
n		1	2	3	4	5	6	7	8
symboles		Kio	Mio	Gio	Tio	Pio	Eio	Zio	Yio
préfixes binaires		kibi	mébi	gibi	tébi	pébi	exbi	zebi	yobi

TABLE 3.4 – Préfixes multiplicateurs en décimal et en binaire

Ainsi, le préfixe p prendra la forme $p = f^n$ où f est le facteur multiplicatif qui vaut 10^3 dans le système décimal et 2^{10} dans le système binaire, et n la puissance recherchée ($n = 1$ pour kilo ou kibi, $n = 2$ pour méga ou mébi...): **prefixe = facteur**n**.

3.9.5 Résultat

On applique la méthode précédente aux giga-octets (qu'on devrait appeler des gibi-octets) : $n = 3$ et $f = 2^{10}$.

Compte-tenu de ces valeurs, le code PYTHON ci-contre permet de calculer le nombre d'octets dans un giga-octet.

Listing 3.8 – giga-octets ou gibi-octets

```

1 n = 3
2 facteur = 2**10
3 prefixe = facteur**n

```

Remarque : on n'a pas cherché à effectuer « à la main » les calculs numériques : PYTHON les fera mieux que nous ; et surtout, on n'a pas cherché non plus à particulariser la 3^{ème} ligne du code en **prefixe = (2**10)**3** : la forme plus abstraite **prefixe = facteur**n** restera identique pour calculer aussi bien en décimal qu'en binaire des yotta-octets ou des yobi-octets, seules les valeurs de **facteur** et de **n** changeront (lignes 1 et 2 du code).

3.9.6 Vérification

Pour vérifier le résultat précédent, on pourra se référer au site officiel de la **Commission Electrotechnique Internationale**. On y vérifie bien que 1 Gio = 1 073 741 824 octets alors que 1 Go = 1 000 000 000 octets. Dans la pratique, on parle toujours de kilo-, méga- ou giga-octets tout en faisant référence aux kibi-, mébi ou gibi-octets.

```
>>> n = 3
>>> facteur = 2**10
>>> prefixe = facteur**n
>>> prefixe
1073741824
```

```
>>> n = 3
>>> facteur = 10**3
>>> prefixe = facteur**n
>>> prefixe
1000000000
```

Une autre vérification nécessaire concerne la généralité de la méthode : on doit pouvoir l'appliquer à d'autres exemples de préfixes binaires. C'est l'objet de la section 3.9.7 suivante.

3.9.7 Généralité

Pour tester la généralité de la méthode précédente, calculer, à l'aide du code PYTHON précédent, les valeurs en octets des préfixes suivants :

- | | |
|---------|---------|
| 1. kibi | 3. tébi |
| 2. mébi | 4. pébi |

3.9.8 Entraînement

Dans le même esprit, proposer une instruction de type « affectation » qui permettra de déterminer les différences entre les préfixes décimaux et binaires correspondants.

- | | |
|--------------|---------------|
| 1. kilo-kibi | 4. téra-tébi |
| 2. méga-mébi | 5. péta-pébi |
| 3. giga-gibi | 6. yotta-yobi |

3.10 Retours d'expériences

3.10.1 Algorithmes

3.10.2 Mrv : méthode

Etant donné un énoncé qui propose de résoudre un exercice portant sur un cas particulier donné, la première étape de la démarche MRV consiste à décrire une méthode générique qui, lorsqu'on l'appliquera, permettra de résoudre le cas particulier considéré ainsi que tout problème équivalent à celui qui est posé.

Pour expliciter une telle méthode générique, on a commencé par s'abstraire des données spécifiques, en particulier numériques et alphanumériques, pour ne considérer que les variables associées. Puis, on a exprimé la solution s recherchée sous la forme d'une fonction f des données x_1, x_2, \dots, x_n abstraites du problème : $s = f(x_1, x_2, \dots, x_n)$.

L'association systématique de noms à des valeurs constitue ainsi un premier moyen d'abstraction qui permet de ne pas faire appel aux données spécifiques à un problème particulier dans l'élaboration de la méthode générique.

3.10.3 Mrv : résultat

Etant donné une méthode générique pour résoudre un ensemble de problèmes équivalents, la deuxième étape de la démarche MRV consiste à appliquer cette méthode à un problème particulier.

Ayant fait abstraction des valeurs particulières dans l'explicitation de la méthode générique, appliquer la méthode a consisté ici à attribuer ces valeurs particulières aux différentes variables

qui interviennent dans la détermination de la solution recherchée. En effet, dans les exercices précédents sur l'affectation, la solution \mathbf{s} recherchée a toujours pris la forme $\mathbf{s} = \mathbf{f}(x_1, x_2, \dots, x_n)$. La fonction \mathbf{f} ayant été déterminée par la première étape de la démarche MRV, il ne reste plus qu'à donner des valeurs aux variables x_1, x_2, \dots pour être capable d'évaluer le membre de droite $\mathbf{f}(x_1, x_2, \dots, x_n)$ de l'affectation et donc la solution \mathbf{s} , membre de gauche de cette affectation.

On passe ainsi d'un cas particulier à un autre en changeant simplement les valeurs des données spécifiques aux problèmes particuliers.

3.10.4 Mrv : vérification

Vérification du résultat : Etant donné un résultat obtenu par application d'une méthode générique pour résoudre un problème particulier, la troisième étape de la démarche MRV consiste à vérifier ce résultat par des méthodes alternatives ou complémentaires de celle déjà utilisée.

Les exemples précédents ont mis en œuvre tout un éventail de méthodes pour vérifier les résultats obtenus :

- des méthodes alternatives qui calculent la solution par une autre voie comme le calcul du produit scalaire par une méthode géométrique (3.7.6) ou le calcul de distance par un logiciel éprouvé : le module `turtle` de PYTHON (3.6.6) ;
- la reconstruction des données à partir des résultats obtenus comme pour le calcul du quotient et du reste de la division entière (3.5.6) ;
- des jeux de tests comme pour le lancer de dés (3.3.6) ou la conversion d'unités (3.8.6) ;
- l'estimation d'un ordre de grandeur comme dans le prix d'un livre (3.2.6) ou de simples calculs « à la main » lorsque c'est possible comme pour les chaînes de caractères (3.4.6) ;
- la consultation d'un ouvrage de référence reconnu comme dans le cas des préfixes binaires (3.9.6).

L'application de ces différentes méthodes de vérification ne prouve pas formellement que les résultats obtenus sont corrects, mais renforce ou non la confiance que l'on peut en avoir. L'une ou l'autre de ces méthodes pourra être reprise, avec d'autres qui apparaîtront au fur et à mesure des chapitres suivants, pour vérifier systématiquement les résultats des exercices. Le cas particulier de la méthode par jeux de tests sera même systématiquement intégré dans la spécification des fonctions (partie III, chapitre 7).

Vérification de la méthode : Il reste à tester la généralité de la méthode en l'appliquant à des exemples différents relevant du même type de problème. C'est ce qui a été systématiquement réalisé pour chacun des exemples traités (3.2.7, 3.3.7, 3.4.7, 3.5.7, 3.6.7, 3.7.7, 3.8.7, 3.9.7).

Chapitre 4

Tests

4.1 Rappels de cours

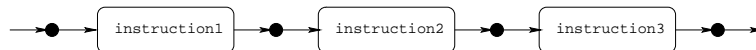


FIGURE 4.1 – Flux séquentiel d'instructions

Sauf mention explicite, les instructions d'un algorithme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites. Le « chemin » suivi à travers un algorithme est appelé le flux d'instructions (Figure 4.1), et les constructions qui le modifient sont appelées des instructions de contrôle de flux. On exécute normalement les instructions de la première à la dernière, sauf lorsqu'on rencontre une instruction de contrôle de flux : de telles instructions vont permettre de suivre différents chemins suivant les circonstances. C'est en particulier le cas de l'instruction conditionnelle qui n'exécute une instruction que sous certaines conditions préalables. On distingue ici 3 variantes d'instructions conditionnelles (Table 4.1) :

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

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

TABLE 4.1 – Instructions conditionnelles en PYTHON

4.1.1 Test simple

L'instruction « **if** » sous sa forme la plus simple permet de tester la validité d'une condition. Si la condition est vraie, alors le bloc d'instructions **blocIf** après le « **:** » est exécuté. Si la condition est fausse, on passe à l'instruction suivante dans le flux d'instructions.

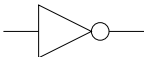

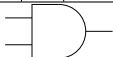

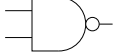
Définition 3 (test simple). *Le test simple est une instruction de contrôle du flux d'instructions qui permet d'exécuter une instruction sous condition préalable.*

La condition évaluée après l'instruction « **if** » est donc une expression booléenne qui prend soit la valeur **False** (faux) soit la valeur **True** (vrai). Elle peut contenir les opérateurs de comparaison suivants (Table 4.2) :

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

TABLE 4.2 – Opérateurs de comparaison en PYTHON

Mais certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme d'une simple comparaison. Par exemple, la condition $x \in [0, 1[$ s'exprime par la combinaison de deux conditions $x \geq 0$ et $x < 1$ qui doivent être vérifiées en même temps. Pour combiner ces conditions, on utilise les opérateurs logiques **not**, **and** et **or** (Table 4.3). Ainsi la condition $x \in [0, 1[$ pourra s'écrire en PYTHON : **(x >= 0) and (x < 1)**.

opérateurs booléens																																						
négation	disjonction	conjonction																																				
<p>not a</p> <table><tr><th>a</th><th>\bar{a}</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table> 	a	\bar{a}	0	1	1	0	<p>a or b</p> <table><tr><th>a</th><th>b</th><th>$a + b$</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> 	a	b	$a + b$	0	0	0	0	1	1	1	0	1	1	1	1	<p>a and b</p> <table><tr><th>a</th><th>b</th><th>$a \cdot b$</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> 	a	b	$a \cdot b$	0	0	0	0	1	0	1	0	0	1	1	1
a	\bar{a}																																					
0	1																																					
1	0																																					
a	b	$a + b$																																				
0	0	0																																				
0	1	1																																				
1	0	1																																				
1	1	1																																				
a	b	$a \cdot b$																																				
0	0	0																																				
0	1	0																																				
1	0	0																																				
1	1	1																																				
	<p>not (a or b)</p> 	<p>not (a and b)</p> 																																				

$\forall a, b, c \in \{0; 1\}$:

$$a + 0 = a \quad a \cdot 1 = a$$

$$a + 1 = 1 \quad a \cdot 0 = 0$$

$$a + a = a \quad a \cdot a = a$$

$$a + \bar{a} = 1 \quad a \cdot \bar{a} = 0$$

$$a + (a \cdot b) = a \quad a \cdot (a + b) = a$$

$$\bar{\bar{a}} = a \quad \overline{a + b} = \bar{a} \cdot \bar{b} \quad \overline{a \cdot b} = \bar{a} + \bar{b}$$

$$(a + b) = (b + a) \quad (a \cdot b) = (b \cdot a)$$

$$(a + b) + c = a + (b + c)$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

TABLE 4.3 – Opérateurs logiques en PYTHON

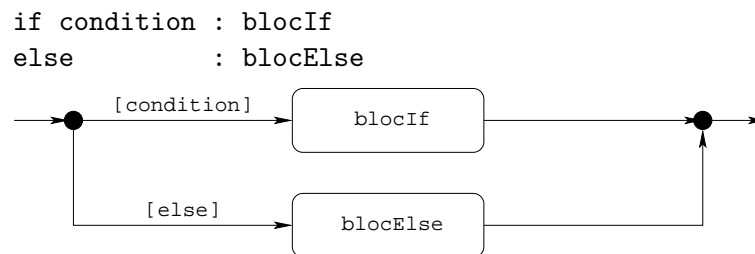
4.1.2 Alternative simple

L'instruction « **if ... else** » teste une condition. Si la condition est vraie, alors le bloc d'instructions **blocIf** après le « **:** » est exécuté. Si la condition est fausse, c'est le bloc

d'instructions `blocElse` après le « `else :` » (sinon) qui est exécuté. Seul l'un des 2 blocs est donc exécuté.

Définition 4 (alternative simple). *L'alternative simple est une instruction de contrôle du flux d'instructions qui permet de choisir entre deux instructions selon qu'une condition est vérifiée ou non.*

La figure 4.2 montre qu'une alternative se comporte comme un aiguillage de chemin de fer dans le flux d'instructions. Un « `if ... else` » ouvre deux voies correspondant à deux traitements différents, et seule une de ces voies sera empruntée (un seul des deux traitements est exécuté). A ce titre, le test simple de la section 4.1.1 précédente est équivalent à une alternative simple où on explicite le fait de ne rien faire (instruction `pass`) dans le bloc d'instructions associé au `else`.



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

FIGURE 4.2 – Alternative simple en PYTHON

Mais il y a des situations où deux voies ne suffisent pas : on utilise alors des alternatives simples en cascade (ou alternatives multiples).

4.1.3 Alternative multiple

L'instruction « `if ... elif` » teste une première condition. Si cette condition est vraie, alors le bloc d'instructions `blocIf` est exécuté. Si la première condition est fausse, on teste la deuxième (`condition1`). Si la deuxième condition est vérifiée, c'est le bloc d'instructions `blocElif1` après le premier « `elif :` » (sinon-si) qui est exécuté ; sinon on teste la condition suivante (`condition2`). Si elle est vérifiée, c'est le bloc d'instructions `blocElif2` après le deuxième « `elif :` » qui est exécuté et ainsi de suite. Si aucune des conditions n'est vérifiée, c'est le bloc d'instructions `blocElse` qui est exécuté. Dans tous les cas, un seul des blocs est donc exécuté.

Définition 5 (alternative multiple). *L'alternative multiple est une instruction de contrôle du flux d'instructions qui permet de choisir entre plusieurs instructions en cascade d'alternatives simples.*

4.2 Vie courante : mention au baccalauréat

4.2.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : déterminer la mention au bac.

4.2.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

```
if condition : bloc1  
else          : bloc2
```

alternative multiple

```
if   condition1 : bloc1  
elif condition2 : bloc2  
elif condition3 : bloc3  
...  
else            : blocn
```

4.2.3 Enoncé

4.2.4 Méthode

4.2.5 Résultat

4.2.6 Vérification

4.2.7 Généricité

4.2.8 Entraînement

4.3 Jeux : 421

4.3.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : .

4.3.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

```
if condition : bloc1  
else          : bloc2
```

alternative multiple

```
if   condition1 : bloc1  
elif condition2 : bloc2  
elif condition3 : bloc3  
...  
else            : blocn
```

4.3.3 Enoncé

4.3.4 Méthode

4.3.5 Résultat

4.3.6 Vérification

4.3.7 Généricité

4.3.8 Entraînement

4.4 Textes :

4.4.1 Objectif

Mettre en œuvre l'instruction d'alternative multiple sur un exemple de type « textes ».

4.4.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

```
if condition : bloc1  
else       : bloc2
```

alternative multiple

```
if   condition1 : bloc1  
elif condition2 : bloc2  
elif condition3 : bloc3  
...  
else           : blocn
```

4.4.3 Enoncé

4.4.4 Méthode

4.4.5 Résultat

4.4.6 Vérification

4.4.7 Généricité

4.4.8 Entraînement

4.5 Nombres :

4.5.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : .

4.5.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

```
if condition : bloc1  
else       : bloc2
```

alternative multiple

```
if   condition1 : bloc1  
elif condition2 : bloc2  
elif condition3 : bloc3  
...  
else           : blocn
```

4.5.3 Enoncé

4.5.4 Méthode

4.5.5 Résultat

4.5.6 Vérification

4.5.7 Généricité

4.5.8 Entraînement

4.6 Figures :

4.6.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : .

4.6.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

```
if condition : bloc1
else          : bloc2
```

alternative multiple

```
if condition1 : bloc1
elif condition2 : bloc2
elif condition3 : bloc3
...
else          : blocn
```

4.6.3 Enoncé

4.6.4 Méthode

4.6.5 Résultat

4.6.6 Vérification

4.6.7 Généricité

4.6.8 Entraînement

4.7 Mathématiques : graphe de fonction

4.7.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : .

4.7.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

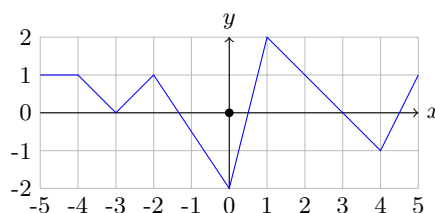
```
if condition : bloc1
else          : bloc2
```

alternative multiple

```
if condition1 : bloc1
elif condition2 : bloc2
elif condition3 : bloc3
...
else          : blocn
```

4.7.3 Enoncé

On considère dans \mathbb{R} la fonction continue f , affine par morceaux, définie sur $[-5; 5]$ par le graphe ci-dessous et $\forall x < -5, f(x) = f(-5)$ et $\forall x > 5, f(x) = f(5)$.



Proposer une instruction de type « alternative multiple » qui permettra de calculer la fonction $y = f(x) \forall x \in \mathbb{R}$.

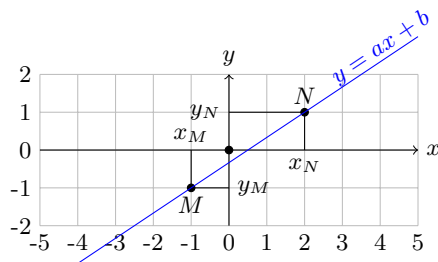
4.7.4 Méthode

Il s'agit de déterminer la valeur $y = f(x)$ d'une fonction continue affine par morceaux sur \mathbb{R} . L'axe des réels $] -\infty, x_1, x_2, \dots, x_n, +\infty[$ est donc vu comme une succession d'intervalles $] -\infty, x_1[, [x_1, x_2[, \dots, [x_{n-1}, x_n[$ et $[x_n, +\infty[$ sur lesquels la fonction f est définie respectivement par les fonctions f_1, f_2, \dots, f_n et f_{n+1} :

$$\begin{aligned} y = f(x) &= f_1(x) & \forall x \in] -\infty, x_1[\\ &= f_2(x) & \forall x \in [x_1, x_2[\\ &= f_3(x) & \forall x \in [x_2, x_3[\\ &= \dots \\ &= f_n(x) & \forall x \in [x_{n-1}, x_n[\\ &= f_{n+1}(x) & \forall x \in [x_n, +\infty[\end{aligned}$$

Chacune des fonctions f_i correspond à une droite d'équation $y = a_i x + b_i$ où a_i représente la pente de la droite et b_i son ordonnée à l'origine. Lorsqu'on connaît 2 points $M(x_M, y_M)$ et $N(x_N, y_N)$ d'une droite d'équation $y = ax + b$, les coefficients a (pente de la droite) et b (ordonnée à l'origine) de la droite sont obtenus par résolution du système de 2 équations : $y_M = ax_M + b$ et $y_N = ax_N + b$. On obtient alors a et b :

$$a = \frac{y_N - y_M}{x_N - x_M} \text{ et } b = \frac{y_M x_N - y_N x_M}{x_N - x_M}$$



Pour la droite ci-contre :

$$a = \frac{1 - (-1)}{2 - (-1)} = \frac{2}{3} \text{ et } b = \frac{(-1) \cdot 2 - 1 \cdot (-1)}{2 - (-1)} = -\frac{1}{3}$$

On vérifie graphiquement ces résultats : pour passer de M à N , on se déplace de $\Delta x = 3$ horizontalement puis de $\Delta y = 2$ verticalement (d'où la pente $a = \Delta y / \Delta x = 2/3$), et la droite coupe bien l'axe des ordonnées en $y = -1/3$.

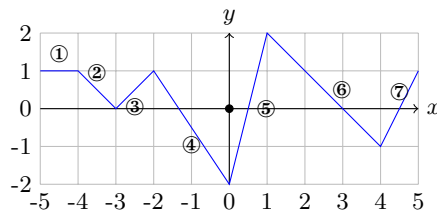
Une fois déterminés les coefficients a_i et b_i de chaque droite, on détermine la valeur de la fonction $y = f(x)$ par une alternative multiple du genre :

```

if x < x1 : y = a1*x + b1
elif x < x2 : y = a2*x + b2
elif x < x3 : y = a3*x + b3
...
elif x < xn : y = an*x + bn
else : y = an+1*x + bn+1
    
```

4.7.5 Résultat

On applique la méthode précédente à la fonction f de l'énoncé. Il faut donc déterminer les équations de droite correspondant aux différents segments du graphe de la fonction, à savoir :



- ① $y = 1$
- ② $y = -x - 3$
- ③ $y = x + 3$
- ④ $y = -3x/2 - 2$
- ⑤ $y = 4x - 2$
- ⑥ $y = -x + 3$
- ⑦ $y = 2x - 9$

Compte-tenu de ces équations, le code ci-contre permet de calculer $y = f(x)$, y compris pour $x < -5$ ($y = f(-5) = 1$) et $x > 5$ ($y = f(5) = 1$).

Remarque : on aurait pu simplifier les deux premières lignes de ce code en

```
if x < -4 : y = 1
```

car les instructions associées sont identiques (ie. les fonctions affines sont identiques sur $] -\infty, -5[$ et $[-5, -4[$).

Listing 4.1 – graphe d'une fonction

```
1 if x < -5 : y = 1
2 elif x < -4 : y = 1
3 elif x < -3 : y = -x - 3
4 elif x < -2 : y = x + 3
5 elif x < 0 : y = -3*x/2 - 2
6 elif x < 1 : y = 4*x - 2
7 elif x < 4 : y = -x + 3
8 elif x < 5 : y = 2*x - 9
9 else : y = 1
```

4.7.6 Vérification

Pour tester le résultat précédent, on peut comparer les valeurs obtenues par le calcul avec celles lues directement sur le graphe pour quelques points caractéristiques. Ces points de mesure sont choisis judicieusement : ils ne correspondent pas aux bornes des intervalles déjà prises en compte dans la méthode mais plutôt à des points où la fonction s'annule (exemples : $x = -4/3$, $1/2$, 3 ou $9/2$) ou à des points d'abscisses aux nœuds de la grille de lecture (exemples : $x = -1$ ou $x = 2$). On peut vérifier par exemple pour $x = -1$ ($y = f(-1) = -1/2$) et $x = 3$ ($y = f(3) = 0$).

```
>>> x = -1
>>> if x < -4 : y = 1
>>> elif x < -3 : y = -x - 3
>>> elif x < -2 : y = x + 3
>>> elif x < 0 : y = -3*x/2 - 2
>>> elif x < 1 : y = 4*x - 2
>>> elif x < 4 : y = -x + 3
>>> elif x < 5 : y = 2*x - 9
>>> else : y = 1
```

```
>>> y
-0.5
```

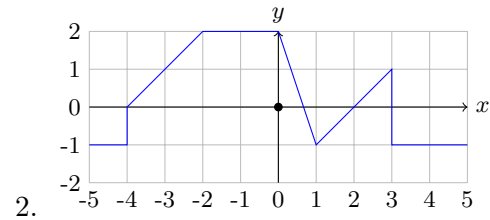
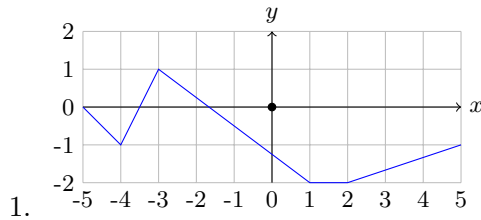
```
>>> x = 3
>>> if x < -4 : y = 1
>>> elif x < -3 : y = -x - 3
>>> elif x < -2 : y = x + 3
>>> elif x < 0 : y = -3*x/2 - 2
>>> elif x < 1 : y = 4*x - 2
>>> elif x < 4 : y = -x + 3
>>> elif x < 5 : y = 2*x - 9
>>> else : y = 1
```

```
>>> y
0
```

On obtient bien par le calcul les résultats lus sur la grille.

4.7.7 Généricité

Pour vérifier la généricité de la méthode précédente, proposer une alternative multiple pour chacune des 2 fonctions définies sur $[-5; 5]$ par les graphes ci-dessous et $\forall x < -5, f(x) = f(-5)$ et $\forall x > 5, f(x) = f(5)$.



4.7.8 Entraînement

Utiliser une alternative multiple pour déterminer les racines réelles d'un trinôme du second degré $ax^2 + bx + c$ à coefficients a , b et c réels.

4.8 Physique : états de l'eau

4.8.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : .

4.8.2 Syntaxe Python

4.8.3 Enoncé

4.8.4 Méthode

4.8.5 Résultat

4.8.6 Vérification

4.8.7 Généricité

4.8.8 Entraînement

4.9 Informatique :

4.9.1 Objectif

Principal : mettre en œuvre l'instruction d'alternative multiple.

Secondaire : .

4.9.2 Syntaxe Python

test simple

```
if condition : bloc
```

alternative simple

```
if condition : bloc1
else          : bloc2
```

alternative multiple

```
if condition1 : bloc1
elif condition2 : bloc2
elif condition3 : bloc3
...
else           : blocn
```

4.9.3 Enoncé

4.9.4 Méthode

4.9.5 Résultat

4.9.6 Vérification

4.9.7 Généricité

4.9.8 Entraînement

4.10 Retours d'expériences

4.10.1 Méthode

4.10.2 Résultat

4.10.3 Vérification

Chapitre 5

Boucles

5.1 Rappels de cours

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

De nouvelles instructions de contrôle de flux sont introduites pour répondre à ces questions : les instructions itératives. On parle également de boucles, de répétitions ou encore d'itérations. Nous distinguerons 2 variantes d'instructions itératives (Table 5.1) : l'itération conditionnelle (**while**) et le parcours de séquences (**for**).

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

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

TABLE 5.1 – Instructions itératives en PYTHON

A propos des instructions itératives, on parle souvent des boucles « **while** » ou des boucles « **for** » dans le jargon des informaticiens.

5.1.1 Itération conditionnelle

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

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

Définition 6 (itération conditionnelle). *L'itération conditionnelle est une instruction de contrôle du flux d'instructions qui permet sous condition préalable de répéter zéro ou plusieurs fois la même instruction.*

Dans une itération conditionnelle, la condition doit évoluer au cours des différents passages dans la boucle afin de pouvoir sortir de la boucle. En ce qui concerne le nombre de passages dans la boucle, deux cas extrêmes peuvent se produire :

- la condition n'est pas vérifiée la première fois : on ne passe alors jamais dans la boucle.
 Exemple : $x = 4$ x est positif ; la condition $x < 0$ n'est donc pas vérifiée
 $y = 0$ la première fois : on ne rentre pas dans la boucle.
 $\text{while } x < 0 : y = y + x$
- la condition n'est jamais faussée : on ne sort jamais de la boucle ; on dit qu'on a affaire à une boucle « sans fin ».
 Exemple : $x = 4$ y est initialement nul : on rentre dans la boucle ;
 $y = 0$ l'instruction du corps de la boucle ne peut
 $\text{while } y \geq 0 : y = y + x$ qu'incrémenter y puisque x est positif : y sera donc
toujours positif et on ne sortira jamais de la boucle.

Le cas de la boucle « sans fin » est évidemment dû le plus souvent à une erreur involontaire qu'il faut savoir détecter assez vite pour éviter un programme qui « tourne » indéfiniment sans s'arrêter.

Dans tous les cas, que l'on connaisse ou non *a priori* le nombre de passages dans la boucle, on peut toujours utiliser l'itération conditionnelle (boucle **while**) pour répéter plusieurs fois un bloc d'instructions à condition de connaître la condition d'arrêt pour sortir de la boucle.

- Lorsqu'on connaît *a priori* le nombre de passages dans la boucle, il suffit de définir un compteur qui compte le nombre de fois où on passe dans la boucle. On initialise correctement ce compteur avant la boucle, on incrémente le compteur dans le corps de la boucle et on sort de la boucle lorsque ce compteur dépasse le nombre de fois connu où on doit passer dans la boucle.
- Lorsqu'on ne connaît pas *a priori* le nombre de passages dans la boucle, il faut absolument déterminer la condition d'arrêt de l'algorithme. Il faut également s'assurer que cette condition sera bien atteinte au bout d'un certain nombre de passages dans la boucle.

5.1.2 Parcours de séquences

Il est fréquent de manipuler des suites ordonnées d'éléments comme les chaînes de caractères (exemple : $s = "123"$), les tableaux (exemple : $t = [1, 2, 3]$) et les n-uplets (exemple : $u = 1, 2, 3$). Chaque élément d'une séquence est accessible par son rang dans la séquence grâce à l'opérateur « crochets » : **sequence[rang]** (exemples : $s[1]$, $t[2]$ ou $u[0]$) et par convention, le premier élément d'une séquence a le rang 0 (exemples : $s[1]$ est le 2^{ème} élément de la chaîne s , $t[2]$ le 3^{ème} élément du tableau t et $u[0]$ le 1^{er} élément du n-uplet u).

>>> $s = "123"$	>>> $t = [1, 2, 3]$	>>> $u = 1, 2, 3$
>>> $s[1]$	>>> $t[2]$	>>> $u[0]$
'2'	3	1

Définition 7 (séquence). *Une séquence est une suite ordonnée d'éléments, éventuellement vide, accessibles par leur rang dans la séquence.*

Les principales opérations sur les séquences sont listées dans le tableau ci-dessous

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

La dernière fonction de ce tableau crée un tableau d'entiers compris entre `start` inclus (= 0 par défaut) et `end` exclus par pas de `step` (= 1 par défaut).

```

>>> range(3)           >>> s = "bonjour"           >>> u1 = 10,12,14
[0, 1, 2]              >>> range(len(s))         >>> u2 = 'a','b','c'
>>> range(3,9,2)       [0, 1, 2, 3, 4, 5, 6]    >>> range(len(u1+u2))
[3, 5, 7]              >>> t = [4,2,6,5,3]       [0, 1, 2, 3, 4, 5]
>>> range(7,0,-1)      >>> range(max(t),min(t),-1) >>> range(len(2*u2[0:2]))
[7, 6, 5, 4, 3, 2, 1] [6, 5, 4, 3]             [0, 1, 2, 3]

```

Il existe une instruction de contrôle adaptée au parcours de séquence :

```
for element in sequence : blocFor
```

```

équivalente à : i = 0
                while i < len(s):
                    element = sequence[i]
                    blocFor
                    i = i + 1

```

5.1.3 Imbrications de boucles

De la même manière que l'on peut cascader des alternatives simples (voir section ??), on peut encapsuler une boucle dans une autre boucle.

Les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point (:), suivie d'une ou de plusieurs instructions indentées (décalées à droite) sous cette ligne d'en-tête (figure ??).

```

ligne d'en-tête:
    première instruction du bloc
    ...
    dernière instruction du bloc

```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, elles doivent l'être exactement au même niveau (décalage de 4 caractères espace, par exemple). Ces instructions indentées

constituent ce qu'on appellera désormais un bloc d'instructions. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction « `while i < 10 :` » constituent un même bloc logique : ces deux lignes ne sont exécutées – toutes les deux – que si la condition testée avec l'instruction `while` est vérifiée, c'est-à-dire si le multiplicateur `i` est tel que $1 \leq i < 10$.

5.1.4 Exécutions de boucles

La maîtrise de l'algorithmique requiert deux qualités complémentaires [?] :

- il faut avoir une certaine intuition, car aucun algorithme ne permet de savoir *a priori* quelles instructions permettront d'obtenir le résultat recherché. C'est là qu'intervient la forme « d'intelligence » requise pour l'algorithmique : la « créativité » de l'informaticien. Il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, les réflexes, cela s'acquiert (en particulier, l'annexe ?? page ?? présente une méthode pour aider à construire des boucles). Et ce qu'on appelle l'intuition n'est finalement que de l'expérience accumulée, tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».
- il faut être méthodique et rigoureux. En effet, chaque fois qu'on écrit une série d'instructions que l'on croit justes, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter (sur papier ou dans sa tête) afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas d'intuition. Mais elle reste néanmoins indispensable si l'on ne veut pas écrire des algorithmes à l'« aveuglette ». Et petit à petit, à force de pratique, on fera de plus en plus souvent l'économie de cette dernière étape : l'expérience fera qu'on « verra » le résultat produit par les instructions, au fur et à mesure qu'on les écrira. Naturellement, cet apprentissage est long, et demande des heures de travail patient. Aussi, dans un premier temps, il faut éviter de sauter les étapes : la vérification méthodique, pas à pas, de chacun des algorithmes représente plus de la moitié du travail à accomplir...et le gage de progrès.

Pour améliorer la compréhension d'une boucle, on peut « tracer » son exécution de tête, à la main ou par programme. Dans tous les cas, l'idée est de suivre pas à pas l'évolution des variables qui interviennent dans la boucle : on détermine leur valeur juste avant la boucle, à la fin de chaque itération et juste après la boucle.

5.1.5 Construction d'une boucle

Un algorithme est un mécanisme qui fait passer un « système » d'une « situation » dite initiale (ou précondition) à une « situation » finale (postcondition ou but). Le couple (situation initiale, situation finale) est appelé spécification de l'algorithme. L'algorithmique vise donc à construire rationnellement des algorithmes à partir de leur spécification.

Le raisonnement qui permet de passer d'une compréhension intuitive d'un tel énoncé à l'algorithme n'est pas toujours facile à concevoir d'un coup. Dans le cas d'une boucle on pourra systématiser la conception de l'algorithme autour de 4 étapes (d'après [?] et [?]) :

1. **Invariant** (ou hypothèse de récurrence) : « Le clou est planté dans la planche ».
2. **Condition d'arrêt** : « La tête touche la planche ».
3. **Progression** : « Frapper un coup de marteau de façon à enfoncer un peu plus le clou ».
4. **Initialisation** : « Planter légèrement le clou à la main ».

Il faut noter que les étapes 1 et 2 définissent des situations tandis que les étapes 3 et 4 concernent des actions. Dans cette section, on notera les situations entre crochets (`[]`) pour les distinguer des actions.

- La recherche d'un invariant est l'étape clé autour de laquelle s'articule la conception des boucles. La conjonction de l'invariant et de la condition d'arrêt conduit logiquement au but recherché :

```
[« invariant » and « condition d'arrêt »] ⇒ [« postcondition »]
```

La condition d'arrêt seule n'implique pas le but.

- La progression doit :
 - conserver l'invariant. Plus précisément, la progression est un fragment d'algorithme défini par les situations initiale et finale suivantes :
 - situation initiale : `[« invariant » and not « condition d'arrêt »]`
 - situation finale : `[« invariant »]`

```
[« invariant » and not « condition d'arrêt »]
« progression »
[« invariant »]
```

- faire effectivement progresser vers le but pour faire en sorte que la condition d'arrêt soit atteinte au bout d'un temps fini.
- L'initialisation doit instaurer l'invariant. Plus précisément, elle doit, partant de la précondition, atteindre l'invariant.

```
[« précondition »]
« initialisation »
[« invariant »]
```

D'une manière plus générale, les 4 étapes de construction d'une boucle sont les suivantes :

1. **Invariant** : proposer une situation générale décrivant le problème posé (hypothèse de récurrence). C'est cette étape qui est la plus délicate car elle exige de faire preuve d'imagination.
2. **Condition d'arrêt** : à partir de la situation générale imaginée en [1], on doit formuler la condition qui permet d'affirmer que l'algorithme a terminé son travail. La situation dans laquelle il se trouve alors est appelée situation finale. La condition d'arrêt fait sortir de la boucle.
3. **Progression** : se « rapprocher » de la situation finale, tout en faisant le nécessaire pour conserver à chaque étape une situation générale analogue à celle retenue en [1]. La progression conserve l'invariant.
4. **Initialisation** : initialiser les variables introduites dans l'invariant pour que celui-ci soit vérifié avant d'entrer dans la boucle. L'initialisation « instaure » l'invariant.
5. **Boucle finale** : Une fois les 4 étapes précédentes menées à leur terme, l'algorithme recherché aura la structure finale suivante (figure ??) :

```
[« précondition »]
« initialisation »
[« invariant »]
while not [« condition d'arrêt »] :
    « progression »
    [« invariant »]
[« postcondition »]
```

Quand on sort de la boucle, la situation finale attendue est atteinte.
Dans la pratique, on ne garde que les instructions :

```
« initialisation »  
while [not « condition d'arrêt »] :  
    « progression »
```

Exemple de la puissance ?? :

```
k = 1  
p = x  
while not k > n :  
    p = p*x  
    k = k + 1
```

Exemple du pgcd ?? :

```
while not b == 0 :  
    r = a%b  
    a = b  
    b = r
```

Un des problèmes, pour l'apprenti informaticien, est que la boucle finale ainsi obtenue ne fait pas apparaître explicitement l'invariant dans le code. L'invariant est une aide conceptuelle pour construire la boucle, mais pas pour l'exécuter.

Définition 8. *Un invariant de boucle est une propriété vérifiée tout au long de l'exécution de la boucle.*

Cette façon de procéder permet de « prouver » la validité de l'algorithme au fur et à mesure de son élaboration. En effet la situation générale choisie en [1] est en fait l'invariant qui caractérise la boucle `while`. Cette situation est satisfaite au départ grâce à l'initialisation de l'étape [4] ; elle reste vraie à chaque itération (étape [3]). Ainsi lorsque la condition d'arrêt (étape [2]) est atteinte cette situation nous permet d'affirmer que le problème est résolu. C'est également en analysant l'étape [3] qu'on peut prouver la terminaison de l'algorithme.

5.2 Vie courante : dépiler des assiettes

5.2.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : dépiler des assiettes d'une pile d'assiettes.

5.2.2 Syntaxe Python

5.2.3 Enoncé

5.2.4 Méthode

5.2.5 Résultat

5.2.6 Vérification

5.2.7 Généricité

5.2.8 Entraînement

5.3 Jeux : rechercher une carte

5.3.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : rechercher une carte donnée dans un jeu de cartes.

5.3.2 Syntaxe Python

5.3.3 Énoncé

5.3.4 Méthode

5.3.5 Résultat

5.3.6 Vérification

5.3.7 Généricité

5.3.8 Entraînement

5.4 Textes : compter les voyelles

5.4.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : compter le nombre de voyelles dans une chaîne de caractères.

5.4.2 Syntaxe Python

5.4.3 Énoncé

5.4.4 Méthode

5.4.5 Résultat

5.4.6 Vérification

5.4.7 Généricité

5.4.8 Entraînement

5.5 Nombres : conversion décimal \rightarrow binaire

5.5.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : convertir un entier décimal en un entier binaire.

5.5.2 Syntaxe Python

5.5.3 Énoncé

5.5.4 Méthode

5.5.5 Résultat

5.5.6 Vérification

5.5.7 Généricité

5.5.8 Entraînement

5.6 Figures : tracé d'un heptagone régulier

5.6.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : tracer un heptagone régulier.

5.6.2 Syntaxe Python

5.6.3 Énoncé

5.6.4 Méthode

5.6.5 Résultat

5.6.6 Vérification

5.6.7 Généricité

5.6.8 Entraînement

5.7 Mathématiques : intégration de $\cos(x)$

5.7.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : calculer l'intégrale de la fonction cosinus.

5.7.2 Syntaxe Python

5.7.3 Enoncé

5.7.4 Méthode

5.7.5 Résultat

5.7.6 Vérification

5.7.7 Généricité

5.7.8 Entraînement

5.8 Physique : sorties d'un circuit logique

5.8.1 Objectif

Principal : mettre en œuvre une instruction d'itération.

Secondaire : déterminer la table de vérité d'un circuit logique.

5.8.2 Syntaxe Python

5.8.3 Enoncé

5.8.4 Méthode

5.8.5 Résultat

5.8.6 Vérification

5.8.7 Généricité

5.8.8 Entraînement

5.9 Informatique :

5.9.1 Objectif

Principal : mettre en œuvre l'instruction d'itération.

Secondaire : .

5.9.2 Syntaxe Python

5.9.3 Enoncé

5.9.4 Méthode

5.9.5 Résultat

5.9.6 Vérification

5.9.7 Généricité

5.9.8 Entraînement

5.10 Retours d'expériences

5.10.1 Méthode

5.10.2 Résultat

5.10.3 Vérification

Chapitre 6

Instructions imbriquées

6.1 Rappels de cours

6.2 Vie courante

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : .

6.2.1 Objectif

6.2.2 Syntaxe Python

6.2.3 Enoncé

6.2.4 Méthode

6.2.5 Résultat

6.2.6 Vérification

6.2.7 Généricité

6.2.8 Entraînement

6.3 Jeux : drapeau tricolore

6.3.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : .

6.3.2 Syntaxe Python

6.3.3 Enoncé

6.3.4 Méthode

6.3.5 Résultat

6.3.6 Vérification

6.3.7 Généricité

6.3.8 Entraînement

6.4 Textes : recherche d'un motif

6.4.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : recherche une chaîne de caractères au sein d'une autre chaîne.

6.4.2 Syntaxe Python

6.4.3 Enoncé

6.4.4 Méthode

6.4.5 Résultat

6.4.6 Vérification

6.4.7 Généricité

6.4.8 Entraînement

6.5 Nombres : crible d'Eratostène

6.5.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : déterminer les n premiers nombres premiers.

6.5.2 Syntaxe Python

6.5.3 Enoncé

6.5.4 Méthode

6.5.5 Résultat

6.5.6 Vérification

6.5.7 Généricité

6.5.8 Entraînement

6.6 Figures :

6.6.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : .

6.6.2 Syntaxe Python

6.6.3 Enoncé

6.6.4 Méthode

6.6.5 Résultat

6.6.6 Vérification

6.6.7 Généricité

6.6.8 Entraînement

6.7 Mathématiques : zéro d'une fonction

6.7.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : .

6.7.2 Syntaxe Python

6.7.3 Enoncé

6.7.4 Méthode

6.7.5 Résultat

6.7.6 Vérification

6.7.7 Généricité

6.7.8 Entraînement

6.8 Physique :

6.8.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : .

6.8.2 Syntaxe Python

6.8.3 Enoncé

6.8.4 Méthode

6.8.5 Résultat

6.8.6 Vérification

6.8.7 Généricité

6.8.8 Entraînement

6.9 Informatique :

6.9.1 Objectif

Principal : mettre en œuvre les instructions de base : affectation, tests, boucles.

Secondaire : .

6.9.2 Syntaxe Python

6.9.3 Enoncé

6.9.4 Méthode

6.9.5 Résultat

6.9.6 Vérification

6.9.7 Généricité

6.9.8 Entraînement

6.10 Retours d'expériences

6.10.1 Algorithmes

6.10.2 Mrv : méthode

6.10.3 Mrv : résultat

6.10.4 Mrv : vérification

Troisième partie

Fonctions

Chapitre 7

Spécification

Chapitre 8

Appels de fonctions

Chapitre 9

Récurtivité

Quatrième partie

Tout en un

Chapitre 10

Vie courante : recherche d'un chemin

Chapitre 11

Jeux : sudoku

Chapitre 12

Textes : cryptographie

Chapitre 13

Nombres : calculateur en base b

Chapitre 14

Figures : construction de figures

Chapitre 15

Mathématiques : systèmes linéaires

Chapitre 16

Physique : équation différentielle

Chapitre 17

Informatique : machine de Turing