

1. Introduction et premières instructions (cours)

L'informatique a pour objectif de réaliser des **traitements automatiques sur les données**. Les programmes informatiques ne font que calculer et transformer des données. Ces données peuvent être très diverses :

- des nombres,
- des textes,
- du son,
- des images ...

Les données que traitent les programmes informatiques peuvent être extrêmement complexes. Mais elles se construisent toutes à partir de quelques briques de bases. On distingue principalement les données numériques des données textuelles.

Les traitements automatiques sur les données peuvent également être très complexes. Il s'agit alors de concevoir des méthodes spécifiques traduites en algorithmes puis en programmes informatique.

1.1 Algorithmes et programmation : principaux concepts

1.1.1 Résolution de problème en quatre phases

Phase 1. L'**analyse**, qui permet de poser le problème à résoudre (données en entrées, données en sorties, calculs à effectuer...)

Exemple 1 :

Problème : définir une billetterie automatique permettant d'acheter des billets de train.

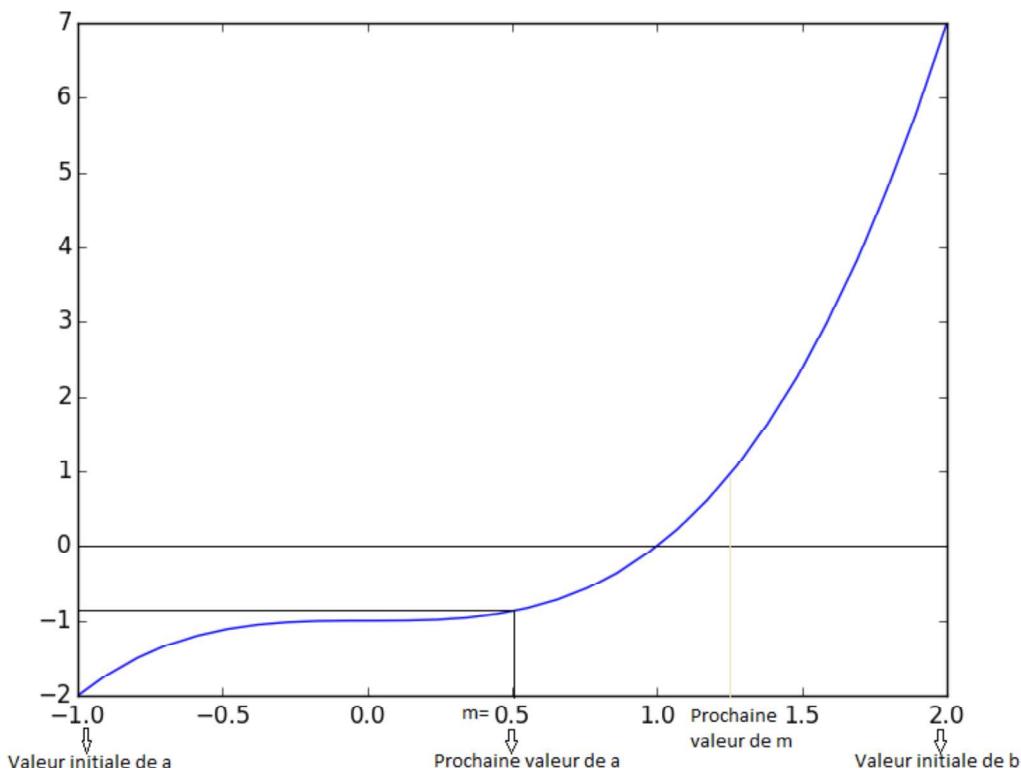
Analyse : il faut prévoir au moins quatre étapes : (1) un dialogue avec le client qui permet de connaître les villes de départ et d'arrivée ainsi que la date du voyage et le nombre de places ; (2) une interaction avec la centrale de réservation pour savoir s'il reste des places disponibles et à quelles conditions tarifaires ; (3) l'étape de paiement avec le calcul du prix du voyage (en fonction des éventuels tarifs préférentiels) et une connexion avec la banque du client pour permettre le paiement ; (4) de nouveau une connexion avec la centrale de réservation pour bloquer les billets achetés et pour les fournir aux clients (déclencher un envoi postal, une impression ou autres).

Exemple 2 :

Problème :

Analyse : Méthode dichotomique de résolution (cf Figure 1) :

- on calcule m le milieu de $[a; b]$, si $f(m) = 0$ alors on a trouvé la solution ;
- si $f(m)$ et $f(b)$ sont de même signe, c'est que la solution se trouve dans $[a; m]$: on affecte à b la valeur de m afin de pouvoir continuer le processus ;
- dans le cas contraire, la solution se trouve dans $[m; b]$: on affecte à a la valeur de m afin de pouvoir continuer le processus ;

FIGURE 1 – $f(x) = x^3 - 1$

- on recommence le processus jusqu'à obtenir un encadrement de m suffisamment petit, c'est-à-dire $|b - a| \leq \epsilon$.

Chacune des étapes constitue une action complexe qu'il faut décomposer en une séquence finie d'opérations élémentaires.

Phase 2. La conception d'un **algorithme**, qui décompose une méthode de résolution en une séquence finie d'opérations élémentaires.

Définition d'un **algorithme** : un algorithme est une suite finie d'opérations non ambiguës prenant éventuellement une ou des données en entrée et fournissant une ou des données en sortie. Les algorithmes manipulent des données provenant de sources différentes : utilisateurs (saisie au clavier), bases de données, web...

Pour décrire un algorithme, on utilise un pseudo-code (ou pseudo-langage ou langage algorithmique) qui utilise des mots du langage naturel. Ce n'est pas un langage informatique. Nous suivrons néanmoins une norme stricte.

Schéma général de présentation d'un algorithme :

Algo

Déclaration de variables

```

Begin
|   Instructions
End

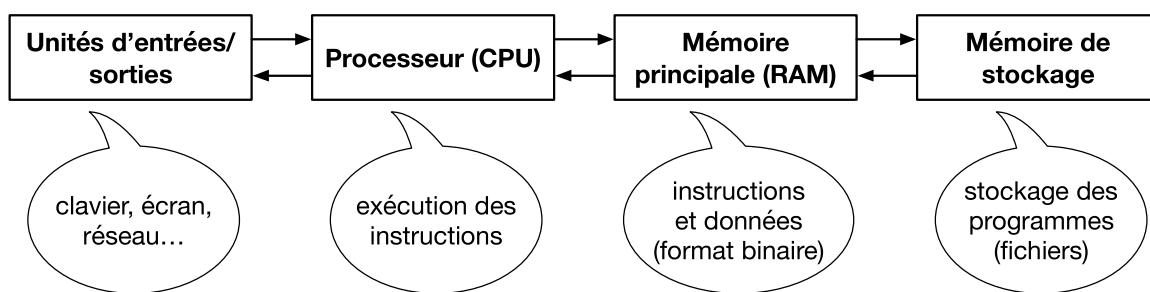
```

Phase 3. La traduction de l'algorithme en **programme informatique** qui se fait à l'aide d'un **langage informatique**. Un programme informatique constitue la traduction d'un algorithme dans un langage informatique choisi : la syntaxe précise (vocabulaire + règles de grammaire) doit alors être respectée (une tabulation mal placée et le programme informatique ne « tourne » pas ou, pire, ne fait pas ce qu'il est censé faire).

Phase 4. **Exécution** du programme sur un ordinateur. En Python, l'exécution se déroule de la façon suivante : les instructions sont lues par l'interpréteur Python qui les exécute les unes à la suite des autres.

1.1.2 Modèle d'ordinateur

Un programme est la traduction d'un algorithme dans un langage de programmation compréhensible par l'ordinateur. En informatique, une **instruction** est une opération élémentaire qu'un programme demande à un processeur d'effectuer. C'est l'ordre le plus basique que peut comprendre un ordinateur. On peut modéliser de manière simplifiée un ordinateur :



Un ordinateur est une machine électronique composée principalement d'une mémoire et d'un processeur. Le rôle de ce dernier est de lire une séquence d'instructions, appelée programme, et d'exécuter, pour chaque instruction, l'opération correspondante (calcul arithmétique, transfert de donnée...).

La mémoire principale (aussi appelée RAM – *Random Access Memory*) sert, pendant l'exécution d'un programme, à stocker les instructions à exécuter et les données associées. Les informations contenues dans cette mémoire (instructions et données) disparaissent lorsque l'alimentation électrique est coupée ; il faut donc prévoir un second support de mémorisation, permanent cette fois, si l'on veut conserver les instructions afin de ré-exécuter le programme. C'est le rôle de la mémoire de stockage (ou mémoire secondaire) classiquement composée de disques durs ou de mémoire flash (disque SSD, clés USB).

L'exécution d'un programme se déroule donc ainsi :

1. le programme est tout d'abord amené en mémoire principale à partir d'un fichier se trouvant sur un support de stockage (disque, clé USB, DVD...). Cette étape est normalement effectuée automatiquement par le système d'exploitation (Windows, MacOS, Linux...) lorsque l'utilisateur lance le programme, par exemple par un double-clic ;
2. le processeur exécute les instructions se trouvant en mémoire principale, une par une, dans l'ordre imposé par le programme, jusqu'à exécution de la dernière instruction.

Un processeur est dénué d'intelligence : il exécute parfaitement les instructions mais n'en comprend pas le sens ; il est également incapable de détecter tout seul des incohérences dans les données, il faudra inclure une vérification explicite.

- Un programme devra spécifier intégralement et sans ambiguïté les tâches à effectuer. Écrire un programme revient donc à expliquer en détail à une machine ce qu'elle doit faire, et ceci dans un langage très précis (au niveau des mots et de la syntaxe).
- Tous les cas devront être envisagés et prévus dans le programme (cas standards, cas particuliers, cas d'erreur...).
- On ne peut pas compter sur le processeur pour nous signaler des incohérences non prévues par le programme.

1.2 Valeurs et types

Un programme informatique manipule des **valeurs**. Ces valeurs sont codées dans la mémoire d'un ordinateur par des suites de 0 et de 1. Il est donc nécessaire d'attribuer un **type** à ces suites de 0 et de 1 pour les interpréter correctement. Par exemple, s'il est de type entier, le code 01001111 représente 79 et, s'il est de type caractère, ce même code représente le caractère 'O' (il s'agit du code ASCII correspondant). Le type détermine donc un ensemble de valeurs et détermine également les opérations que l'on peut effectuer sur ces valeurs.

Nous allons considérer pour le moment les types de base : `int`, `float` et `str`. Le type `int` pour l'ensemble des nombres entiers, le type `float` pour celui des nombres décimaux et le type `str` (pour `string`) pour les chaînes de caractères.

Sur les données de type `int` et `float`, on peut appliquer les opérateurs arithmétiques usuels (listés par priorité croissante) :

- négation (changement de signe) ;
- + et - addition et soustraction ;
- * et / multiplication et division ;
- `div` et `mod` division entière et reste de la division entière¹ ;
- `**` puissance.

Une valeur de type `str` est une suite de caractères encadrée par des guillemets comme par exemple "Nom :". Un caractère peut être une lettre (majuscule ou minuscule), un chiffre, un symbole de ponctuation, une espace, et bien d'autres choses encore. Nous introduirons au fur et à mesure des types supplémentaires.

En Python, le type `float` désigne les nombres flottants, ou plus précisément à virgule flottante. Il s'agit des nombres comprenant une partie décimale après la virgule, celle-ci est marquée par un point conformément à l'usage anglo-saxon : 10.2 par exemple. Python accepte aussi la notation scientifique : 6.7e-4 pour la valeur 0.00067 ($= 6.7 \times 10^{-4}$), ou 6.7e5 pour la valeur 670000.0 ($= 6.7 \times 10^5$) par exemple.

Le type d'une valeur est inféré (déduit) automatiquement par Python. On parle de **typage dynamique**. Une valeur composée de chiffres et ne comportant pas de . est un `int` ; si elle comporte un point, c'est un `float`.

Les opérateurs Python sur les données de type `int` et `float` sont (listés par priorité croissante) :

1. Étant donné deux nombres entiers a et b , si on note q le quotient de la division entière de a par b et r son reste, on peut écrire : $a = q * b + r$

- négation (changement de signe) ;
- + et - addition et soustraction ;
- * et / multiplication et division ;
- // et % division entière et reste de la division entière ;
- ** puissance.

Remarque à propos de la division entière Si on souhaite effectuer une division entière d'un nombre a par un nombre b , on écrira en python : $a // b$ pour obtenir le quotient et $a \% b$ pour obtenir le reste.

Sur un ordinateur, tout calcul effectué sur des nombres entiers est garanti exact. En revanche, les calculs effectués sur des flottants peuvent être entachés d'erreurs.

En voici un exemple, si on réalise le calcul suivant

$1.1 + 2.2 - 3.3$

on n'obtient pas 0.0 mais un nombre très proche de 0.0 à savoir $4.440892098500626 \times 10^{-16}$.

La différence peut sembler insignifiante, mais combinée à d'autres calculs, elle peut devenir très significative.

En Python, les chaînes de caractères sont écrites en les encadrant soit de deux simples apostrophes ' (appelées quotes), soit de guillemets " (appelés doubles quotes). Ce sont des délimiteurs de chaîne. Vous noterez que la présence des délimiteurs permet de distinguer la valeur numérique 2 du caractère '2' (ou de manière équivalente "2").

Aucune limitation n'existe sur le nombre de caractères que contient une chaîne. Ce nombre est appelé la **longueur** de la chaîne.

Une chaîne de caractères peut ne contenir aucun caractère. On parle alors de la chaîne vide. Littéralement, la chaîne vide s'écrit avec deux quotes ou deux doubles quotes : '' ou "".

Le choix des délimiteurs (' ou ") est en général indifférent. Mais si une chaîne de caractères doit contenir l'un de ces deux symboles, il faut choisir l'autre comme délimiteur.

Introduisons une première opération sur les chaînes de caractères : la **concaténation**. Cette opération consiste à construire une chaîne de caractères à partir de deux chaînes données en les mettant bout à bout. L'opération de concaténation est réalisée par l'opérateur +. Par exemple, la chaîne 'bonjour' peut s'obtenir avec l'opération de concaténation suivante :

'bon' + 'jour'

En Python la fonction type donne le type de la valeur qu'on lui donne entre parenthèses. En voici un exemple :

```
>>> type(100)
<class 'int'>
>>> type(100.)
<class 'float'>
>>> type('100')
<class 'str'>
```

1.3 Variables, affectation et expression

Ces valeurs sont stockées dans des zones mémoire de l'ordinateur. Chaque zone a une adresse mais, dans un programme informatique, on n'accède pas aux valeurs en donnant une adresse mémoire mais en référençant une adresse par une **variable**.

1.3.1 En pseudo-code

En pseudo-code, une variable désigne une boite dans laquelle on range une valeur. Une variable porte un nom : il s'agit de son identificateur. Toutes les variables utilisées dans l'algorithme doivent être déclarées au préalable suivant la syntaxe :

```
identificateur : type
```

L'affectation est l'instruction qui permet de ranger une valeur dans une variable. En pseudo-code, on la notera avec une flèche \leftarrow ce qui signifie « prend la valeur » :

```
identificateur_variable  $\leftarrow$  valeur
```

L'**évaluation** d'une variable consiste à accéder à la valeur qu'elle contient.

Une **expression** est une formule combinant des variables, des valeurs, des opérateurs... Quand une expression est évaluée, elle renvoie un résultat unique et typé. L'évaluation d'une expression complexe (combinant différents opérateurs) se fait avec les priorités usuelles de l'arithmétique, mais l'emploi des parenthèses est VIVEMENT conseillé.

De façon générale, l'affectation s'écrit comme suit en langage algorithmique :

```
identificateur_variable  $\leftarrow$  expression
```

L'affectation se déroule comme suit : l'expression est évaluée et la valeur renvoyée lors de cette évaluation est « rangée » dans la variable. La valeur antérieure éventuelle de la variable *identificateur_variable* est alors écrasée et remplacée par la valeur renvoyée par l'expression.

```
Algo
  x,y : int
Begin
  |   x  $\leftarrow$  6
  |   y  $\leftarrow$  2*x
End
```

Exemple Cet algorithme a pour objet :

1. d'attribuer la valeur 6 à la variable x ;
2. pour la variable y, il s'agit d'abord d'évaluer l'expression $2 \times x$ qui renvoie 12 et d'attribuer cette valeur 12 à la variable y.

1.3.2 En Python

Une variable permet de référencer une zone mémoire (ici la boite du pseudo-code correspond concrètement à une zone mémoire de l'ordinateur). L'affectation est l'instruction qui permet d'associer une variable à une valeur stockée dans une zone mémoire. La syntaxe de l'affectation est :

```
identificateur = expression
```

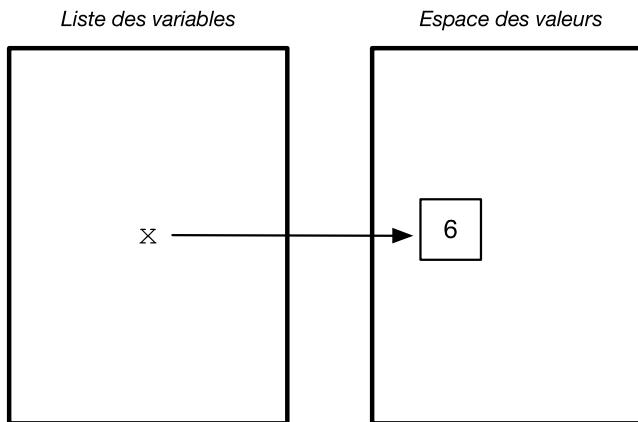
L'évaluation d'une variable consiste à accéder à la valeur qu'elle référence : on dit qu'elle renvoie la valeur contenue dans la zone mémoire.

Contrairement à d'autres langages de programmation, un programme Python ne comporte pas une partie explicite de déclaration de variables : la déclaration d'une variable et son initialisation (c'est-à-dire la première valeur que l'on va stocker dans cette variable) se font en même temps. Une variable n'a pas de type, c'est la valeur qu'elle référence qui est typée !

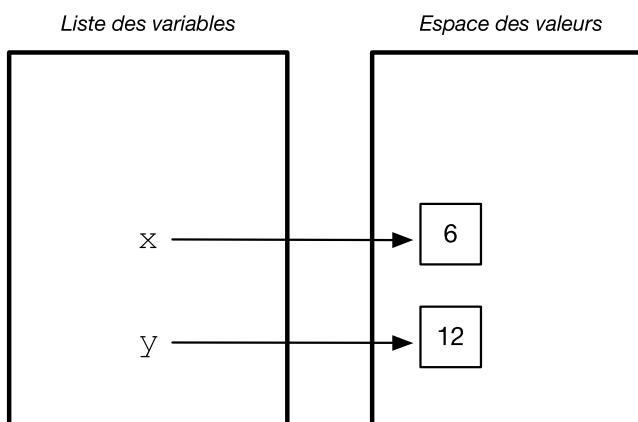
Ainsi l'algorithme du début de section se traduit en Python comme suit :

```
x = 6
y = 2*x
```

À l'issue de la première instruction, une zone mémoire contenant la valeur 6 est créée (dans l'espace des valeurs), son type nombre entier, `int` en Python, est automatiquement déduit par l'interpréteur. Puis, dans la liste des variables, la variable `x` est créée et un lien (une référence) entre `x` et 6 est établi. C'est alors par la variable `x` qu'on peut atteindre la zone mémoire contenant la valeur 6.



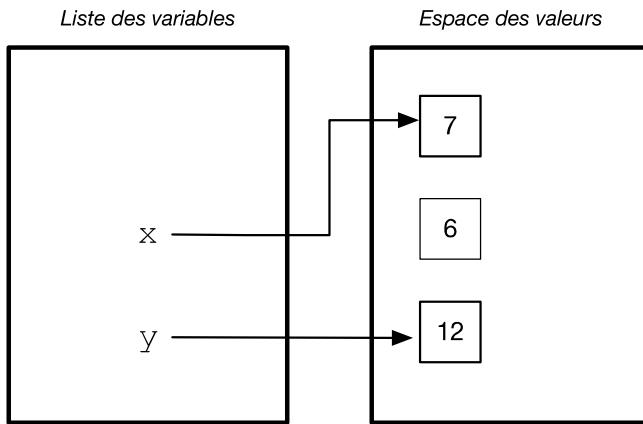
Concernant la deuxième instruction, l'expression `2*x` est en premier lieu évaluée : on accède à la valeur de type `int` référencée par `x` (on dit aussi contenue dans `x`) et on multiplie cette valeur par 2, la valeur renvoyée est alors 12 qui est aussi de type `int` (en multipliant un `int` par un `int` on obtient un `int`!). Dans un deuxième temps, l'espace mémoire contenant 12 va être référencé par la variable `y` qui va donc être créée dans la liste des variables.



Une nouvelle affectation ne change pas le contenu de la case mémoire référencée mais change le lien entre la variable et une nouvelle zone mémoire. Ainsi, si l'on effectue une nouvelle affectation

```
x = 7
```

la variable `x` va référencer une nouvelle case mémoire contenant 7.



Dans une affectation, ce que vous placez à la gauche du signe égal doit donc toujours être une variable, et non une expression, car il s'agit de référencer une valeur (espace des valeurs) par une variable (espace des variables). Ainsi par exemple, l'instruction `x + 1 = y` n'est pas légale et n'a pas de sens en Python.

Attention. Dans une instruction d'affectation, l'opérateur `=` n'est pas ici l'opérateur égalité de l'arithmétique. Si `a = a + 1` est faux en mathématique, cette affectation a un sens en programmation et est très fréquente. L'instruction `a = a + 1` signifie « prend la valeur de `a`, l'augmente de 1 et remet cette valeur dans `a` », ou encore : « incrémenter `a` ».

L'identificateur d'une variable en Python peut être constitué de lettres minuscules (`a` à `z`), de lettres majuscules (`A` à `Z`), de nombres ou du caractère `_`. Mais l'identificateur ne doit pas débuter par un nombre (et éviter de la faire commencer par `_`) et il est conseillé qu'il ne contienne pas de caractère accentué. L'identificateur ne peut pas commencer par des guillemets ou quotes pour ne pas être considéré comme une valeur de type `str`. L'identificateur ne doit pas être choisi parmi les mots réservés du langage. Python est sensible à la casse, ce qui signifie que les variables `nb`, `NB`, `nB` sont différentes. Enfin, on n'utilise jamais d'espace dans un nom de variable puisque l'espace est un séparateur d'instructions. Le choix de l'identificateur est important pour rendre vos instructions le plus compréhensibles possibles. Par exemple, dans le problème de la billetterie automatique, le nom de la variable qui référence la ville de départ pourrait être `v` mais c'est beaucoup plus clair de choisir `ville_depart` comme identificateur.

En plus des opérateurs déjà mentionnés, il existe des opérateurs, dit combinés, qui réalisent à la fois une opération arithmétique et une affectation :

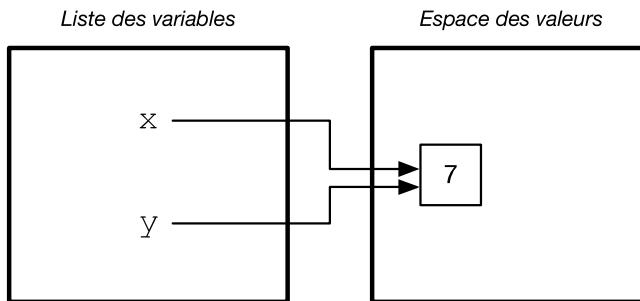
- `a += 2` est équivalent à `a = a + 2`
- `a -= 2` est équivalent à `a = a - 2`
- `a *= 2` est équivalent à `a = a * 2`
- `a /= 2` est équivalent à `a = a / 2`

Type des expressions : si dans une même expression, les opérandes sont de type `int` et `float`, la valeur renvoyée sera de type `float` car ce type est plus général.

En Python, on peut affecter une valeur à plusieurs variables simultanément. Par exemple, l'instruction

```
x = y = 7
```

affecte aux deux variables `x` et `y` la même valeur 7.



On peut aussi effectuer des affectations parallèles à l'aide d'une seule instruction.
Par exemple, l'instruction

```
a, b = 4, 8.33
```

affecte aux variables `a` et `b` les valeurs respectives de 4 et 8,33.

1.3.3 Conversion explicite de type

En Python, on peut aussi faire des **conversions explicites** de type :

```
x = 2.6
y = int(x)
z = float(y) # z vaut 2.0
w = int('2')
```

Après la première affectation, `x` référence la valeur 2.6 de type `float`. Le type `float` est inféré par Python car la valeur est composée de chiffres et contient un point. Après la deuxième affectation, `y` référence la valeur 2 de type `int` : une troncature est réalisée et la partie fractionnaire est supprimée. Après la troisième affectation, `z` référence la valeur de type `float` 2.0. Après la quatrième affectation, `w` référence la valeur de type `int` 2. L'expression `int('2.0')` n'est pas valide : il n'est pas possible de demander deux conversions successives (dont l'une est implicite) : str -> float -> int. Il faut écrire :

```
t = float('2.0')
t = int(t)           ou bien t = int(float('2.0'))
```

Après avoir présenté l'affectation, il s'agit d'expliquer deux autres instructions élémentaires permettant d'interagir avec un utilisateur : **l'instruction d'écriture** pour afficher des données (résultats) à l'écran et **l'instruction de lecture** pour stocker/enregistrer des données en mémoire de l'ordinateur.

1.4 Instruction d'écriture

L'opération d'écriture permet d'afficher une valeur à l'écran.

Pseudo-code

Traduction en Python

```
Algo
    x : type
Begin
    | print x
End
```

```
print(x)
```

En Python, `x` est une expression qui est d'abord évaluée, la valeur renvoyée est ensuite affichée à l'écran. Si c'est une chaîne de caractères, c'est la chaîne qui est affichée (sans que ne soient affichés les délimiteurs de chaînes). Par exemple,

```
print('Hello')
```

La fonction `print()` peut afficher les valeurs de plusieurs expressions passées en argument. Dans ce cas, on sépare les différentes expressions à afficher par une virgule. Par exemple :

```
>>> var = 2
>>> print("2*var=", 2*var)
2*var = 4
```

Par défaut, les affichages des différentes expressions passées en argument sont sur une même ligne séparés par une espace. L'affichage termine par un passage à la ligne (le caractère `\n`). Deux arguments optionnels du `print()` permettent de modifier ce comportement par défaut.

Le premier, `sep`, indique ce qui doit séparer les différentes valeurs affichées. On l'utilise en ajoutant `sep = valeur` à la liste des arguments (après les valeurs à afficher), où `valeur` est de type chaîne de caractères. Par exemple :

```
>>> var = 2
>>> print(var, 2*var, 4*var)
2 4 8
>>> print(var, 2*var, 4*var, sep = ", ")
2,4,8
>>> print(var, 2*var, 4*var, sep = "\n")
2
4
8
```

Le second argument optionnel est `end` qui indique la chaîne à ajouter à la fin de l'affichage (par défaut, il s'agit d'un `\n`, le retour à la ligne). On l'utilise en ajoutant `end = valeur` à la liste des arguments (après les valeurs à afficher), où `valeur` est de type chaîne de caractères. Par exemple :

```
>>> print(var, 2*var, 4*var, end = "Fin")
2 4 8Fin
```

On peut bien évidemment affecter des valeurs aux deux arguments `sep` et `end` dans une même instruction d'écriture :

```
>>> print(var, 2*var, 4*var, sep = ',', end = "Fin")
2,4,8Fin
```

1.5 Instruction de lecture

L'opération de lecture, en pseudo-code **get x**, permet de lire une valeur provenant de l'extérieur (clavier) et de l'affecter à la variable **x**.

En pseudo-code

```
Algo
  x : type
Begin
  | get x
End
```

Traduction en Python

```
x = input()
```

En Python, c'est la fonction `input()` qui permet de créer en mémoire une valeur constituée des caractères saisis au clavier par l'utilisateur. Il suffit alors de référencer cette valeur par une variable pour pouvoir l'utiliser dans un programme.

À l'exécution, l'instruction `input()` provoque l'interruption de l'exécution du programme et attend que l'utilisateur saisisse une valeur au clavier, en terminant par la touche 'entrée' du clavier. Cette valeur est alors enregistrée sous la forme d'une chaîne de caractères et, grâce à l'affectation, cette valeur de type `str` est référencée par la variable `x`.

Par exemple, si on exécute l'instruction `x = input()` et que l'utilisateur saisit au clavier 2022, la valeur référencée par `x` est la chaîne de caractères '`2022`'.

Pour changer le type de la valeur lue par la fonction `input()`, on doit faire une conversion explicite de type. Dans l'exemple ci-dessous, on transforme la valeur lue de type `str` en une valeur de type `int` :

```
x = input()
x = int(x)
```

Dans cet exemple, pour que l'exécution des instructions ne donne pas lieu à une erreur, il faut bien évidemment que l'utilisateur saisisse uniquement des chiffres au clavier. Aucune vérification n'est faite par l'instruction `input()`. Si l'utilisateur saisit autre chose que des chiffres (par exemple `deux`), on aura une erreur à l'exécution (un bug) de la deuxième instruction.

Dans cet exemple, pour convertir en `int` la valeur de type `str` lue par `input()`, on a écrit deux instructions distinctes. Il est possible de réunir ces deux instructions en une seule en appliquant directement la conversion de type sur la sortie du `input()`, sans l'affecter au préalable à `x`. On obtient alors une instruction équivalente :

```
x = int(input())
```

L'exécution de l'instruction `input()` permet d'interrompre l'exécution d'un programme pour permettre à un utilisateur de faire une saisie clavier mais aucune indication n'est donnée à l'utilisateur pour l'informer de ce qu'on attend de lui. Pour l'informer, il faut ajouter une instruction d'écriture préalable (on parle de message d'invite ou prompt). Par exemple :

```
print("Entrez une valeur entière : ")
x = input()
x = int(x)
```

A l'exécution, c'est alors clair pour l'utilisateur : le message

Entrez une valeur entière :

s'affiche. Puis le programme passe à la deuxième instruction et la fonction `input()` est exécutée. Elle interrompt l'exécution du programme pour laisser la main à l'utilisateur qui peut saisir une valeur. Cette valeur est créée par `input()` puis référencée par la variable `x`. La troisième instruction permet alors de convertir le type de cette valeur en `int`.

Il est possible de spécifier un message d'invite dans le `input()` directement (sans passer explicitement par la fonction `print()`). Une autre version du programme précédent est :

```
x = input("Valeur entière : ")
x = int(x)
```

La seule différence est ici que le message "Entrez une valeur entière :" ne sera pas suivi d'un passage à la ligne. L'autre différence est que vous ne pouvez donner qu'une valeur au `input()` alors que vous pouvez en spécifier plusieurs pour le `print()`. On peut écrire :

```
inf = 10
sup = 20
print("Valeur comprise entre", inf, 'et', sup, ':')
x = input()
x = int(x)
```

Alors que l'exécution du programme ci-dessous génère le message d'erreur.

```
inf = 10
sup = 20
x = input("Valeur comprise entre", inf, 'et', sup, ':')
x = int(x)

x = input("Valeur entière comprise entre", inf, 'et', sup, ':')
TypeError: input expected at most 1 argument, got 5
```

On peut néanmoins contourner cette limite en concaténant les chaînes de caractères à écrire. En effet, l'expression

```
'Valeur comprise entre' + str(inf) + 'et' + str(sup) + ':'
```

est valide et, lors de son évaluation, cette expression renvoie une et une seule valeur de type `str`. Il est alors possible d'écrire :

```
inf = 10
sup = 20
x = input('Valeur comprise entre' + str(inf) + 'et' + str(sup) + ':')
x = int(x)
```

Si on combine le message d'invite dans le `input()` et la conversion de type directement sur la valeur renvoyée par le `input()`, on peut passer de la version ci-dessous

```
print("Entrez une valeur entière : ")
x = input()
x = int(x)
```

à cette unique instruction équivalente (on a ajouté un passage à la ligne après le message d'invite) :

```
x = int(input("Entrez une valeur entière : \n"))
```

1.6 Instructions conditionnelles

1.6.1 En pseudo-code

Comme déjà vu, un algorithme permet d'exécuter une suite d'instructions les unes à la suite des autres. Or, on peut aussi vouloir dire dans un algorithme que l'on exécutera certaines instructions uniquement si une certaine condition est vérifiée, ce qui permet d'aiguiller le déroulement d'un programme dans différentes directions et de modifier son comportement. L'**instruction conditionnelle** de base est :

if . . . then . . .

Qui sera en pseudo-code sous la forme :

```
if (Condition) then
|   BlocInstructions
```

où BLOCINSTRUCTIONS est une suite d'instructions qui sera exécutée uniquement si la condition est vérifiée. Si la condition n'est pas vérifiée, alors il ne se passera rien.

Une condition est une expression logique qui fournit un résultat dont le type est `bool`. Le type `bool` est un ensemble de 2 valeurs : `True` ou `False`. Par exemple, $(D < 0)$ renvoie vrai (`True`) lorsque $D = -4$ et renvoie faux (`False`) lorsque $D = 6$.

Les opérateurs de comparaison qui peuvent être utilisés pour écrire les expressions logiques peuvent être : $>$, $<$, $==$, \neq , \leq , \geq .

Attention : pour tester si 2 éléments sont égaux il faut doubler le signe " $=$ " pour ne pas faire à la place une affectation. Nous reviendrons plus en détail dans une section ultérieure sur la définition et la construction d'expressions logiques plus complexes.

L'ensemble des instructions contenues dans :

```
if (Condition) then
|   BlocInstructions
```

constitue une instruction conditionnelle qui commence avec l'évaluation de la condition et se termine à la fin de BLOCINSTRUCTIONS.

Exemple

Voici un algorithme en pseudo-code qui, étant donnée une équation du second degré, détermine si elle n'admet aucune solution réelle.

Rappel : l'équation $ax^2 + bx + c = 0$ n'admet aucune solution réelle si $\Delta = b^2 - 4ac$ est < 0 . L'équation étant entièrement définie par les paramètres a , b et c , ils seront demandés à l'utilisateur et c'est le programme qui compte tenu de ces paramètres calculera Δ et fournira la réponse.

```

Algo
  a : float
  b : float
  c : float
  D : float
Begin
  | print "Quel est le parametre a ?"
  | get a
  | print "Quel est le parametre b ?"
  | get b
  | print "Quel est le parametre c ?"
  | get c
  | D ← (b*b -4*a*c)
  | if (D < 0) then
  |   | print "L'équation n'admet aucune racine reelle"
End

```

Il est également possible d'indiquer en plus à l'algorithme de traiter le cas où la condition n'est pas vérifiée à l'aide du mot clé **else**. La structure de base devient alors :

```

if (Condition) then
  | BlocInstructions1
else
  | BlocInstructions2

```

Le programme exécutera les instructions de **BLOCINSTRUCTIONS1** si le résultat de l'évaluation de la condition est **True** et dans le cas contraire il exécutera les instructions de **BLOCINSTRUCTIONS2**. Comme le résultat est nécessairement dans l'un des 2 cas, le programme exécutera l'un ou l'autre des blocs d'instructions.

Cette structure constitue une nouvelle instruction conditionnelle qui commence avec l'évaluation de la condition et se termine à la fin d'un des **BLOCINSTRUCTIONS**.

Reprendons l'exemple précédent et maintenant dans le cas où Δ est ≥ 0 , le programme affichera qu'il existe au moins une racine réelle.

Algo

```

a : float
b : float
c : float
D : float

Begin
  print "Quel est le parametre a?"
  get a
  print "Quel est le parametre b?"
  get b
  print "Quel est le parametre c?"
  get c
  D ← (b*b -4*a*c)
  if (D < 0) then
    | print "L'équation n'admet aucune racine réelle"
  else
    | print "L'équation a au moins une racine réelle"
End

```

Il est tout à fait possible d'imbriquer des instructions conditionnelles.

Algo

```

a : float

Begin
  print "Quel est le parametre a?"
  get a
  if (a ≠ 0) then
    | if (a mod 2 == 0) then
      | | print "Un nombre pair positif"
      | | print "Un nombre pair"
    | else
      | | print "Il est nul"
End

```

Si on applique l'algorithme précédent en prenant $a = 4$ que se passe-t-il ? et pour $a = 5$?

Il existe une possibilité supplémentaire pour différencier le `else` et introduire des cas intermédiaires, en utilisant le mot clé `else if` :

```

if (Condition 1) then
  | BlocInstructions1
else if (Condition 2) then
  | BlocInstructions2
else if (Condition 3) then
  | BlocInstructions3
  ...
else
  | BlocInstructionsK

```

Si le résultat de CONDITION1 est True alors seul le bloc d'instruction BLOCINSTRUCTIONS1 sera exécuté. Dans le cas contraire (c'est-à-dire si le résultat de CONDITION1 est False) alors le programme va tester la CONDITION2. Si le résultat de CONDITION2 est True, alors seul le bloc d'instruction BLOCINSTRUCTIONS2 sera exécuté. Dans le cas contraire (c'est-à-dire si le résultat de CONDITION2 est False) alors le programme va tester la CONDITION3... Au final, un seul parmi les K blocs d'instructions potentiels sera exécuté et le bloc BLOCINSTRUCTIONS K sera exécuté si le résultat de CONDITION1 est False et le résultat de CONDITION2 est False et le résultat de CONDITION3 est False... et le résultat de CONDITION $K-1$ est False. Il est possible de mettre autant de else if que l'on veut et le mot clé else n'est pas obligatoire à la fin (auquel cas, il peut ne rien se produire); mais s'il est présent, il ne peut y en avoir qu'un.

1.6.2 Traduction en Python

Pour la version if ... then ...

```
if (condition):
    BlocInstructions1
```

Attention

Ne pas oublier le ':' qui délimite le début du bloc et ne pas oublier l'indentation des instructions à exécuter dans le cas où la condition est vérifiée. L'indentation doit être la même pour toutes les instructions du bloc. Notez que les parenthèses autour de la condition ne sont pas obligatoires en Python. Le BlocInstructions1 ne peut pas être vide.

Le type booléen est noté également bool en Python. Aussi les deux valeurs possibles sont True et False (pour vrai et faux).

Attention : True et False sont des mots clé pour Python ; si la majuscule est omise, ils ne sont pas reconnus en tant que tel.

Les opérateurs de comparaison en Python sont : >, <, ==, !=, <=, >=.

Illustration

```
a = float(input("Donner le parametre a : "))
b = float(input("Donner le parametre b : "))
c = float(input("Donner le parametre c : "))
d = b*b - 4*a*c
if d < 0:
    print("Votre équation n'admet aucune racine réelle")
```

Remarque : s'il n'y a qu'une seule instruction à exécuter, alors on peut l'écrire sur la même ligne que les deux points.

Illustration

Que se passe-t-il lors de l'exécution du programme ci-dessous pour $x = 4$?

```
x = int(input("Donner l'inconnue x positive ou nulle : "))
if x >= 10:
    print("x n'est pas constitué d'un unique chiffre")
print("parce qu'il est plus grand que 10")
```

Pour la version if . . . then . . . else . . .

```
if condition:
    BlocInstructions1
else:
    BlocInstructions2
```

Attention : Ne pas oublier les ':' ni l'indentation après else

Illustration

```
a = float(input("Donner le parametre a : "))
b = float(input("Donner le parametre b : "))
c = float(input("Donner le parametre c : "))
d = b*b - 4*a*c
if d < 0:
    print("Votre equation n'admet aucune racine reelle")
else:
    print("Votre equation admet au moins une racine reelle")
```

Question

Que fait le programme Python suivant ?

```
if True:
    print("Tout juste")
else:
    print("Pas bon")
```

Pour la version if . . . then . . . else if . . . else . . .

```
if condition1:
    BlocInstructions1
elif condition2:
    BlocInstructions2
elif condition3:
    BlocInstructions3
...
else:
    BlocInstructionsK
```

Illustration

Dans l'exemple précédent, il est possible d'affiner la réponse en précisant le nombre de racines réelles quand il en existe.

```
a = float(input("Donner le parametre a : "))
b = float(input("Donner le parametre b : "))
c = float(input("Donner le parametre c : "))
d = b*b - 4*a*c
if d < 0:
```

```

print("La valeur du discriminant est négative :", d)
print("L'équation n'admet aucune racine réelle")
elif d == 0:
    print("Le discriminant est nul")
    print("L'équation admet une unique racine réelle")
else:
    print("La valeur du discriminant est positive :", d)
    print("L'équation admet deux racines réelles distinctes")

```

Les instructions du `else` ne seront donc exécutées que si $d \geq 0$ et $d \neq 0$, ce qui correspond bien à $d > 0$.

1.7 Conditions : des expressions logiques

1.7.1 Opérateurs logiques

Il est possible de combiner plusieurs expressions logiques afin d'en obtenir une plus complexe. Les opérateurs définis sur les expressions logiques (en pseudo-code et Python) sont :

- `not` (négation) : en logique souvent notée $\text{not}(E1)$ ou $\overline{E1}$, où $E1$ est une expression logique. Si l'expression $E1$ est `True`, alors le résultat de $\overline{E1}$ est `False` et si le résultat de $E1$ est `False`, alors le résultat de $\overline{E1}$ est `True`. Cela se représente dans ce que l'on appelle une table de vérité qui donne les différents résultats possibles d'une expression logique en fonction des valeurs des expressions logiques la constituant. Pour la négation, la table de vérité associée est :

$E1$	$\overline{E1}$
<code>True</code>	<code>False</code>
<code>False</code>	<code>True</code>

Exemple

Pour D valant -3 , l'expression $D \neq 0$ est `True` et par conséquent l'expression `not($D \neq 0$)` est `False`.

- `or` (disjonction) : en logique souvent notée $E1 \text{ or } E2$ ou $E1 \vee E2$, où $E1$ et $E2$ sont toutes deux des expressions logiques. La table de vérité associée à l'opérateur `or` est :

$E1$	$E2$	$E1 \text{ or } E2$
<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>

Exemple

Soit l'expression $E = (a \neq 100) \text{ or } (b < 0)$. Pour a valant 6 et b valant -10 , E aura la valeur `True` et pour a valant 100 et b valant 7 , E prendra la valeur `False`.

— and (conjonction) : en logique souvent notée $E1 \text{ and } E2$ ou $E1 \wedge E2$, où $E1$ et $E2$ sont toutes deux des expressions logiques. La table de vérité associée à l'opérateur and est :

$E1$	$E2$	$E1 \text{ and } E2$
True	True	True
True	False	False
False	True	False
False	False	False

Exemple

Soit l'expression $E = (a \neq 100) \text{ and } (b < 0)$. Pour a valant 15 et b valant -10 , E aura la valeur True et pour a valant 50 et b valant 20, E prendra la valeur False.

1.7.2 Propriétés

Propriété 1

$$\text{not}(A \text{ or } B) = \text{not}(A) \text{ and } \text{not}(B)$$

Exemple

$\text{not}(x1 \geq 0 \text{ or } x2 == 4)$ est équivalent à $\text{not}(x1 \geq 0) \text{ and } \text{not}(x2 == 4)$ qui est équivalent à l'expression logique : $(x1 < 0) \text{ and } (x2 \neq 4)$.

Démonstration Propriété 1

En utilisant les tables de vérité et en montrant que les tables de vérité de chacune des expressions logiques sont équivalentes.

A	B	$A \text{ or } B$	$\text{not}(A \text{ or } B)$	$\text{not}(A)$	$\text{not}(B)$	$\text{not}(A) \text{ and } \text{not}(B)$
T	T	T	F	F	F	F
T	F	T	F	F	T	F
F	T	T	F	T	F	F
F	F	F	T	T	T	T

Propriété 2

$$\text{not}(A \text{ and } B) = \text{not}(A) \text{ or } \text{not}(B)$$

Exemple

$\text{not}(x3 \neq 99 \text{ and } x4 < 8)$ est équivalent à $\text{not}(x3 \neq 99) \text{ or } \text{not}(x4 < 8)$, équivalent à l'expression logique : $(x3 == 99) \text{ or } (x4 \geq 8)$.

Les propriétés 1 et 2 sont particulièrement utiles pour les tests avec else dans le cas où la condition initiale est du type $(A \text{ and } B)$ ou $(A \text{ or } B)$.

Remarques

La distributivité des opérateurs est la suivante :

$$\begin{aligned} A \text{ and } (B \text{ or } C) &= (A \text{ and } B) \text{ or } (A \text{ and } C) \\ A \text{ or } (B \text{ and } C) &= (A \text{ or } B) \text{ and } (A \text{ or } C) \end{aligned}$$

Sans parenthèse, l'opérateur `and` est prioritaire sur l'opérateur `or` et l'opérateur `not` est prioritaire sur les opérateurs `and` et `or`.

Illustration - Exercice

Dans l'algorithme ci-dessous, le texte de "message2" peut-il être plus explicite quant à la parité et au signe de a ?

```
Algo
  a : float
Begin
  | print "Quel est le parametre a?"
  | get a
  | if (a > 0 and a mod 2 == 0) then
  |   | print "C'est un nombre pair positif"
  | else
  |   | if (a mod 2 == 0) then
  |     |   | print "message2"
End
```

L'utilisation des opérateurs `not`, `and` et `or` permet d'écrire des algorithmes de façon plus simple et en limitant les imbrications des tests. À titre d'illustration, faire exercice ci-dessous.

1. Écrire sans les 3 opérateurs `not`, `and` et `or` un algorithme permettant de dire si une variable a appartient à l'intervalle $[0, 9]$.
2. Même question mais en utilisant l'opérateur `and`
3. Même question mais en utilisant l'opérateur `or`