Chapitre 1-4
Installer son environnement de travail

1. Introduction

Il ne s'agit ici que de CPython, l'implémentation de référence de Python, et non de PyPy ou Jython.
Quel que soit votre système d'exploitation, vous pouvez installer Python en lisant ce chapitre puis, dans un second temps, installer des bibliothèques tierces au gré de vos besoins (cf. section Installer une bibliothèque tierce) et vous pourrez créer des environnements virtuels (cf. section Créer un environnement virtuel).
Si vous souhaitez installer d'un seul coup Python ainsi que Jupyter (anciennement IPython) et la plupart des bibliothèques scientifiques ou d'analyse de données, vous pouvez aller directement à la section Installer Anaconda, pour installer celui-ci en lieu et place de Python. Vous disposerez alors d'autres méthodes pour gérer les environnements virtuels et pour installer des bibliothèques tierces.

2. Installer Python

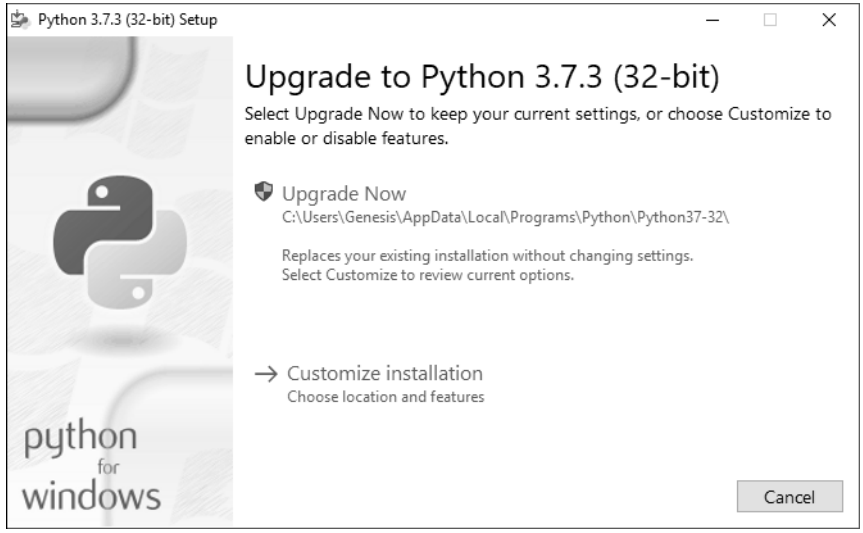
2.1 Pour Windows

Le système d'exploitation Windows requiert usuellement l'utilisation d'un installateur pour pouvoir installer un logiciel quel qu'il soit. Si vous disposez de Windows, vous devriez en avoir l'habitude. Python ne déroge pas à la règle.
Pour installer Python, vous devez donc aller sur le site officiel (https://www.python.org/downloads/) pour télécharger l'installateur adéquat. Comme vous pourrez le constater, on vous met en avant un accès rapide à la dernière version (au moment où ces lignes sont écrites, la 3.11.0), puis un accès aux dernières versions encore actives (actuellement la version 3.10 qui reçoit encore des corrections d'anomalies, puis les versions 3.9 à 3.7 qui reçoivent des corrections de sécurité uniquement).

Il est également possible de télécharger la toute dernière version de la branche 2.7 qui est en fin de vie (elle n'est plus mise à jour), car il existe encore de nombreux projets n'ayant pas encore migré.
Le support correctif dure 2 ans après la première sortie de la version et le support de sécurité dure 5 ans.
Pour notre part, nous vous conseillons la dernière 3.x, mais vous êtes libre d'installer celle que vous souhaitez ou même d'en installer plusieurs suivant vos contraintes, il n'y a pas d'objection à cela.
Une fois le téléchargement effectué, vous devez lancer l'installateur (et éventuellement passer quelques protections de votre système qui vous demande d'accorder votre confiance à cet installateur), pour observer l'écran suivant :

Comme vous pouvez le constater, il est possible de personnaliser l'installation en choisissant le chemin d'installation du logiciel ou en choisissant de ne pas sélectionner quelques fonctionnalités, mais nous ne le conseillons pas.
Nous vous recommandons en revanche de cocher la case **Add python.exe to PATH** afin de configurer la variable PATH du terminal pour rendre Python accessible plus facilement.

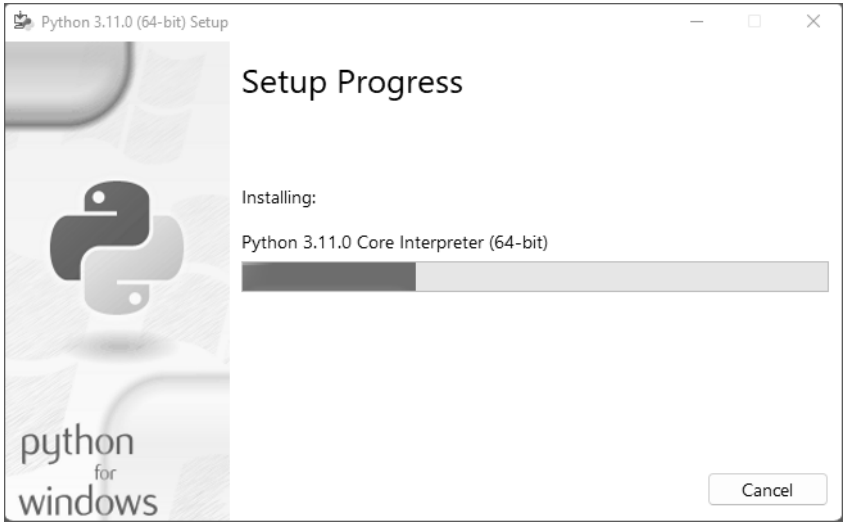
Si vous avez déjà une ancienne version de Python installée de la même branche (dans cet exemple, Python 3.7.2 est déjà installé), vous pourrez la mettre à jour à l'aide du même installeur :



Par contre, si vous avez déjà la version 3.7.1 et que vous installez la version 3.11, cette dernière ne viendra pas remplacer la précédente, mais s'installera à côté. Si vous souhaitez remplacer, il vous faudra donc désinstaller proprement la toute dernière version installée, ce qu'il est possible de faire en relançant l'installeur d'origine.

Nous vous encourageons à garder les installeurs sur votre PC, car ils pourraient devenir indisponibles au téléchargement si trop vieux.

Quel que soit le scénario, vous arriverez devant un écran vous montrant la progression de l'installation et vous n'aurez qu'à fermer la fenêtre une fois celle-ci terminée :



Vous êtes maintenant prêt à utiliser Python.

2.2 Pour Mac

Il faut savoir qu'une version de Python est déjà préinstallée sur Mac, car Mac OS X l'utilise pour ses propres besoins et Python est intégré à son propre cycle de développement. Cependant, si vous souhaitez une version différente de celle qui est déjà présente, vous pouvez l'installer, sachant qu' il n'y a pas de contre-indication à posséder plusieurs versions de Python sur la même machine.

Pour installer Python sur Mac OS X, la procédure à suivre est similaire à celle pour Windows. Il faut donc se rendre sur le site officiel (https://www.python.org/downloads/mac-osx/), télécharger un installateur correspondant à sa configuration et suivre les étapes.

Pour les utilisateurs de Mac, sachez que Python dispose d'une bonne intégration de ses spécificités, en particulier vis-à-vis de Objective-C, le langage de programmation avec lequel est développé Mac OS X, et Cocoa, interface de programmation de Mac OS X.

2.3 Pour GNU/Linux et BSD

Les différentes distributions libres utilisent nativement Python, notamment pour des parties sensibles. Python y est donc tout naturellement déjà installé, généralement sous la dernière version de la branche 2.x. Cependant, ici comme ailleurs, il n'y a pas d'objections à utiliser plusieurs versions de Python.
Le plus simple reste d'utiliser votre gestionnaire de paquets, ce qui peut se faire via un outil graphique, comme Synaptic pour Debian :

Il suffit alors de faire une recherche sur le mot-clé **`python`** pour voir les différentes versions (sur une ancienne Debian, ce sont Python 2.6, 2.7 et 3.2).
Par contre, tous les paquets python3-xxxxx que vous pouvez voir ici sont des bibliothèques tierces et non Python lui-même. Nous en parlerons plus tard dans ce chapitre.
Une fois les paquets souhaités sélectionnés, il ne manque plus qu'à les installer en cliquant sur le bouton **Appliquer**.
Notez que tout ceci peut se faire par la simple ligne de commande, toujours en utilisant votre gestionnaire de paquets qui peut être apt-get, aptitude, yum, emerge, pkg_add ou autre.
Voici par exemple pour une distribution Debian ou Ubuntu :

```
$ sudo aptitude install python3
```

Ceci ne permet cependant pas de choisir la version que l'on souhaite, à moins d'aller trouver des sources alternatives. Si l'on veut avoir la toute dernière version de Python, il faudra la plupart du temps passer par la compilation.

2.4 Par la compilation

Compiler Python n'est pas en soi une tâche très complexe. C'est par contre souvent une tâche imposée lorsque l'on ne travaille pas avec des conteneurs. En effet, en entreprise, on développe souvent des applications qui sont destinées à être hébergées. Il est alors impératif de travailler sur votre propre poste avec une version de Python qui soit la même que celle existante sur la machine de production.

Sous GNU/Linux, mais aussi sous d'autres systèmes, il est possible de compiler la version de Python que l'on souhaite. Après tout, Python n'est rien d'autre qu'un programme écrit en C. Pour ce faire, il faut aller télécharger le code source (https://www.python.org/downloads/source/), qui prend la forme d'une archive, puis décompresser celle-ci, se placer dans le répertoire ainsi obtenu et taper ces quelques commandes :

```
$ ./configure –prefix=/path/to/my/python/directory
$ make
$ sudo make altinstall
```

Notez que dans cette dernière ligne, nous n'utilisons pas la commande **`make install`**, qui aurait pour effet de remplacer votre Python système par le Python que vous compilez, ce qui pourrait avoir des conséquences indésirables voire désastreuses.

Notez également que vous choisissez lors de la configuration le chemin dans lequel vous placerez vos bibliothèques Python. En général, l'usage veut que l'on utilise **`/opt`**, mais il n'y a pas de règle, tout dépend des pratiques de votre entreprise ou votre expérience en la matière.

Si vous venez d'installer Python 3.5 par cette méthode, vous aurez alors maintenant accès à ce programme en l'appelant ainsi, depuis votre terminal :

```
$ python3.5
```

Par cette même méthode, vous pouvez installer les dernières versions (https://www.python.org/download/pre-releases/) de Python qui ne sont pas encore sorties (alphas ou betas), ce qui vous permet de les tester en avant-première !

Notons que, par cette méthode, toutes les bibliothèques de Python ne fonctionneront pas. En effet, lorsqu'elles ont besoin d'autres bibliothèques C, il faut effectuer des compilations croisées et utiliser les différents en-têtes de ces bibliothèques. C'est le cas par exemple pour faire fonctionner Curses, ReportLab (génération de fichiers PDF) ou encore PyUSB (accès aux ports matériels USB).

Dans ce cas-là, la commande **`./configure`** devra recevoir des arguments supplémentaires et vous devrez trouver un tutoriel en ligne pour vous indiquer la démarche, laquelle peut être plus ou moins complexe.

2.5 Pour un smartphone

Installer une machine virtuelle Python sur un smartphone est possible. Pour Android, la procédure est assez simple puisqu'il existe un produit dédié (http://qpython.com/), tout comme sur Windows Phone (https://apps.microsoft.com/store/detail/python-39/9P7QFQMJRFP7). Pour iOS, c'est une autre paire de manches (https://github.com/linusyang/python-for-ios) étant donné que l'utilisateur est enfermé dans un système sur lequel il n'a aucun contrôle.

3. Installer une bibliothèque tierce

Remarque

Si vous abhorrez le terminal, sachez que vous pouvez installer une bibliothèque tierce depuis votre IDE, ce qui sera probablement plus aisé pour vous.

3.1 À partir de Python 3.4

Pour installer une bibliothèque tierce, vous devez simplement connaître son nom. Celui-ci est généralement assez intuitif. Par exemple, la bibliothèque permettant de communiquer avec un serveur Redis s'appelle redis.
Il peut y avoir des variations. Par exemple, la bibliothèque de référence pour traiter du XML est lxml et, plus complexe, celle pour BeautifulSoup est bs4. En recherchant comment répondre à un besoin sur le Net ou sur PyPi (https://pypi.python.org/pypi), vous trouverez rapidement une bibliothèque de référence.
Sur des sujets plus confidentiels, il vous arrivera de trouver plusieurs petites bibliothèques. Vous pourrez alors les tester et choisir celle que vous utiliserez pour votre projet.
Sachez que vous pouvez aussi conduire une recherche directement depuis votre terminal :

```
$ pip search xml
$ pip search soup
```

Cela vous donnera une liste de bibliothèques accompagnée d'une courte description, à la manière de ce que font les gestionnaires de paquets sous Linux (lesquels sont écrits en Python, au passage).
Sachez que **`pip`** existe quel que soit votre système d'exploitation (vous devez être familier avec le terminal de votre système, cependant) et que depuis la version 3.4 de Python, il est installé automatiquement avec celui-ci. Si ce n'est pas votre cas, consultez la section suivante : Pour une version inférieure à Python 3.4.
`pip` est un outil formidable. Si vous utilisez une version de Python qui est celle du système, vous utiliserez alors la commande **`pip`** pour gérer les bibliothèques. Si vous utilisez une autre version, telle que Python 3.5, alors vous utiliserez la commande **`pip-3.5`**. Pour Python 3.3, ce sera **`pip-3.3`**. Dans les exemples suivants, il vous faudra prendre en compte cette particularité.
Cet outil vous permettra d'installer une bibliothèque à sa dernière version ainsi que toutes les bibliothèques dépendantes. En effet, il n'est pas rare qu'une bibliothèque de Python ait besoin d'une autre bibliothèque (ou de plusieurs) pour fonctionner. Par exemple, l'installation de redis se fait par cette commande :

```
$ pip install redis
```

On peut aussi choisir la version à installer :

```
$ pip install -Iv redis==2.10.5
```

Ou mettre à jour la bibliothèque à une version précise :

```
$ pip install -U redis==2.10.5
```

Ou à la dernière version :

```
$ pip install -U redis
```

Et on peut la désinstaller :

```
$ pip uninstall redis
```

Une fonctionnalité très importante permet d'obtenir la liste des bibliothèques déjà installées (quelle que soit la manière dont elles ont été installées) :

```
$ pip freeze
```

Ce que l'on peut mettre dans un fichier :

```
$ pip freeze > requirements.txt
```

Pour installer tous les paquets ainsi listés, il faut procéder ainsi :

```
$ pip install -r requirements/base.txt
```

Cette méthode est particulièrement utile dans le cadre d'un environnement virtuel ; nous y reviendrons.
Il est possible de retrouver des informations sur un paquet déjà installé :

```
$ pip show django-redis
---
Name: django-redis
Version: 4.3.0
Location: /path/to/my/env/lib/python3.4/site-packages
Requires: redis
```

On voit ici que le paquet **`django-redis`** a une dépendance vers **`redis`** : en l'installant, on installe automatiquement **`redis`**.
Mettre à jour ce paquet met à jour automatiquement les dépendances :

```
$ pip install -U django-redis
```

Si on ne veut pas mettre à jour les dépendances, on peut procéder ainsi :

```
$ pip install -U --no-deps django-redis
```

On peut aussi installer plusieurs bibliothèques en même temps :

```
$ pip install django-redis==4.3.0 bs4 lxml
```

Cette commande installera donc automatiquement redis s'il n'est pas installé, car il est déclaré comme dépendance.
Cette commande a cependant des limites. En effet, si vous installez une bibliothèque tierce qui utilise une bibliothèque C, vous devrez disposer des en-têtes C correspondants (paquets **`dev`** pour Debian ou **`devel`** pour Fedora). Il faut donc avoir un peu de pratique dans ce genre de situation pour savoir déjouer ces pièges.

Chapitre 3
Conditions, tests et booléens

1. Les tests et conditions

1.1 Les conditions sont primordiales

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

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

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

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

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

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

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

1.2 Structures conditionnelles

En algorithmie, il existe deux structures conditionnelles :

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

1.2.1 SI ALORS SINON

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

Remarque

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

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

Voici la syntaxe du SI ALORS :

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

Exemple :

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

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

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

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

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

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

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

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

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

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

Remarque

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

1.2.2 CAS PARMI

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

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

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

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

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

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

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

Regardons maintenant comment combiner plusieurs tests dans une structure conditionnelle.

2. La logique booléenne

2.1 Conditions multiples

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

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

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

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

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

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