Chapitre 1: Introduction à l'algorithmique

Brice Mayag
brice.mayag@dauphine.fr

M1 SIEE

DAUPHINE UNIVERSITÉ PARIS

Sommaire

DAUPHINE UNIVERSITÉ PARIS

Sommaire

DAUPHINE UNIVERSITÉ PARIS

Programme du cours

Les grandes notions

- Les bases de l'algorithmique
- La récursivité et le paradigme *Diviser pour Régner*
- Structures de données basiques et avancées : listes, files, piles, tas, arbres de recherche
- Application au filtrage collaboratif

Évaluation

- Examen

Sommaire

L'algorithmique ?

Définition (informelle)

Un algorithme est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :

- définie de façon rigoureuse et non ambigue ;
- effective (i.e. pouvant être réalisée en un temps fini).

La notion d'algorithme est plus générale que celle de programme (indépendant du langage de programmation utilisé).

Un peu d'histoire

Le mot algorithme vient du nom du mathématicien Al Khwarizmi (820) : introduction de la numérotation décimale et des calculs s'y rapportant.

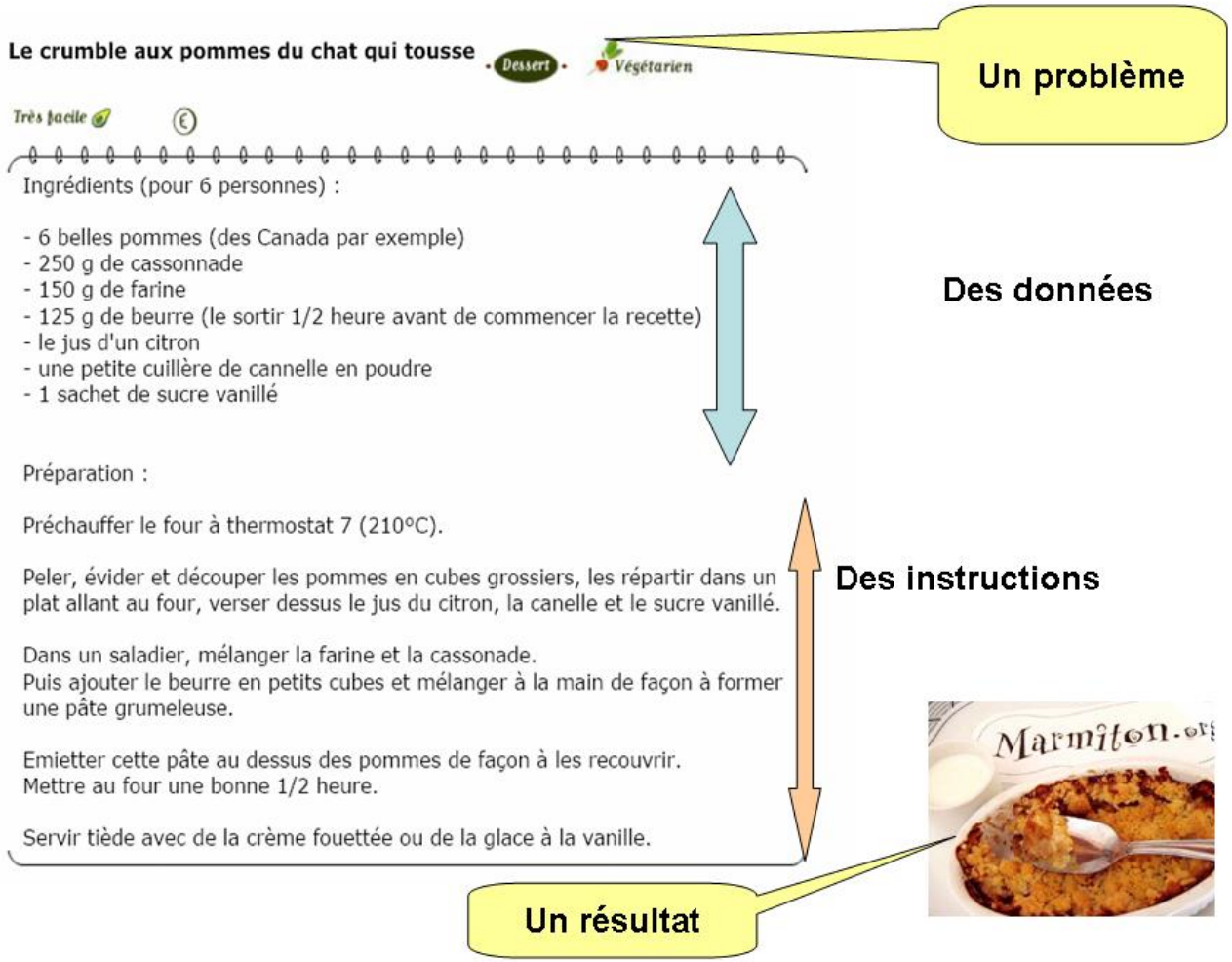
DAUPHINE UNIVERSITÉ PARIS

Des besoins contradictoires

Un algorithme doit :

- être simple à comprendre, à mettre en oeuvre et à mettre au point ;
- mettre intelligement à contribution les ressources de l'ordinateur, et plus précisément, il doit s'exécuter rapidement.

Un algorithme : c'est quoi ?



Classiquement assimilable à une recette de cuisine

Algorithme : définitions

Un algorithme =

- Description précise des opérations à faire pour résoudre un problème (suite d'instructions).
- Procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs et qui donne en sortie une valeur, ou un ensemble de valeurs.

Un *bon* algorithme =

- Un algorithme correct : i.e. pour chaque instance en entrée, l'algorithme se termine en produisant la bonne sortie
  $\Rightarrow$ Savoir prouver un algorithme
- Un algorithme efficace : mesure de la durée que met un algorithme pour produire un résultat
  $\Rightarrow$ Savoir analyser la complexité d'un algorithme : i.e. détermination de l'espace mémoire et du temps d'exécution nécéssaire à la résolution du problème.

Pour un problème donné, plusieurs algorithmes ou aucun sont possibles.
Un algorithme se termine en un temps fini.

DAUPHINE UNIVERSITÉ PARIS

Un exemple simple

Problème de tri

- Données : une suite de n nombres, $(e_0, e_1, ..., e_{n-1})$
- Résultat : permutation de la suite donnée en entrée $(e_0^{'}, e_1^{'}, ..., e_{n-1}^{'})$ de telle sorte que $e_0^{'} \leq e_1^{'} \leq ... \leq e_{n-1}^{'}$

Principe du tri par insertion

On parcourt entièrement la liste en appliquant à chaque itération la stratégie suivante :

- Recherche de la place du ième élément

Tri par insertion

Principe

On veut trier la liste :

3	2	7	1	5

1. On part du constat que liste[0..0] est triée

3	2	7	1	5

2. On met l'élément d'indice 1 à sa place dans la liste[0..1]

2	3	7	1	5

3. liste[0..1] est maintenant triée. On recommence à l'étape 2 avec l'élément suivant.

- Exemple : Trier la liste de nombres

$$[65; 15; 34; 3; 25; 22; 63; 3; 66; 17]$$

Un algorithme simple

Considérons un tableau A contenant une séquence à trier de longueur n

Algorithm 1 TRI-INSERTION (A)

```
1: for j ← 2 to n do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

On verra comment implanter cet algorithme en Python

Analyse d'un algorithme

- Simplicité et intelligibilité de l'algorithme
- Efficacité de l'algorithme :
  - Temps d'exécution
  - Occupation de la mémoire
  - Quantité de trafic généré sur un réseau
  - Quantité de données déplacées sur le disque
  - ...

Complexité d'un algorithme

Permet de quantifier les algorithmes

Deux types de compléxité :

- **En espace** : Quelle quantité de place en mémoire va t-on utiliser ?
- **En temps** : Combien d'opérations va t-on effectuer ?

Intérêt : comparer deux algorithmes

Mesurer le temps d'exécution

- Le temps d'exécution dépend de l'entrée : par exemple dans le cas d'un algorithme de tri, si le tableau est déja trié.
- On cherche une fonction $T(n)$ représentant le **temps d'exécution** d'un algorithme en fonction de la **taille de l'entrée** n (nombre d'élements constituant l'entrée, nombre de bits nécéssaire à la représentation de l'entrée,...)
- Le calcul exact étant souvent impossible à appréhender exactement, on s'intéresse aux :
  - Meilleur des cas
  - Pire de cas
  - Cas moyen : nécéssite d'avoir des connaissances sur la distribution statistique des entrées

Mesurer le temps d'exécution

Cas du tri par insertion

Algorithm 2 TRI-INSERTION (A)

```
1: for j ← 2 to n do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

Soit c_i, le *coût* temporel de chaque instruction. On a alors :
$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$
t_j nombre de fois que le test de la boucle **While** est exécutée pour cette valeur de j

Mesurer le temps d'exécution

Cas du tri par insertion

- **Cas le plus favorable** : le tableau est déja trié et donc pour tout $j = 2..n$ on a $t_j = 1$.
  Temps d'exécution :
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$
  Temps sous la forme $an + b$: **fonction linéaire de n**.
- **Cas le plus défavorable** : la tableau est trié dans l'ordre décroissant donc $t_j = j$ pour tout $j = 2..n$.
  Temps d'exécution : $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n-1)}{2}) + c_6(\frac{n(n-1)}{2}) + c_7(n-1)$
  Temps sous la forme $an^2 + bn + c$: **fonction quadratique de n**

Classes de complexité

- Algorithmes sub-linéaires : $\Theta(\log n)$
- Algorithmes linéaires : $\Theta(n)$ et $\Theta(n \log n)$
- Algorithmes polynomiaux : $\Theta(n^k)$
- Algorithmes exponentiels : $\Theta(2^n)$

Sommaire

Langages de programmation

Ensembles de mots clefs et de règles pour former des phrases pouvant être traduites par la machine (binaire).

Plusieurs niveaux

- Langages de bas niveau : instructions élémentaires très *proches de la machine* (ex : Assembleur : V AX, [0110] signifie *copier le contenu de 0110 dans le registre AX* ; langage machine)
- Langages de haut niveau : instructions plus abstraites (C, C++, Perl, Java, Python).

Paradigmes de programmation

- Procédural (impératif)
- Fonctionnel
- Logique
- Orienté-objet
- Hybride

Paradigmes de programmation

Programmation procédurale

Déclaration de procédures accomplissant des tâches spécifiques nécessaires au programme. Les étapes sont expliquées ligne par ligne, et l'instruction atomique est l'affectation de variables. Le programme est exécuté.

Exemples

FORTRAN,C,PASCAL

Paradigmes de programmation

Programmation fonctionnelle

Langage d'utilisation et de developpement de fonctions. Tout programme est construit par fonctions, retournant un résultat en sortie et prenant en entrée la sortie d'autres fonctions. Le programme est évalué.

- Diviser un problème complexe en sous problèmes
- Récursivité

Exemples

CAML, SCHEME, LISP, ...

Paradigmes de programmation

Programmation logique

Langage déductif utilisant des règles et des faits. Un moteur d'inférence permet de donner le résultat. Le programme est une théorie axiomatique à partir de laquelle on infère (démonstrations logiques/mathématiques) les résultats.

Exemple

PROLOG

Paradigmes de programmation

Couche au-dessus des autres paradigmes de programmation.

Programmation orientée-objet

Basée sur le principe que des choses peuvent avoir des points communs, des similarités en elles-mêmes ou en leur façon d'agir. On conçoit des fabriques (*classes*) qui servent à construire des composants (*objets*). Les classes contiennent des données(*attributs*) et des actions (*méthodes*).

Exemples

C++, JAVA, SMALLTALK, O-CAML...

Paradigmes de programmation

Programmation hybride

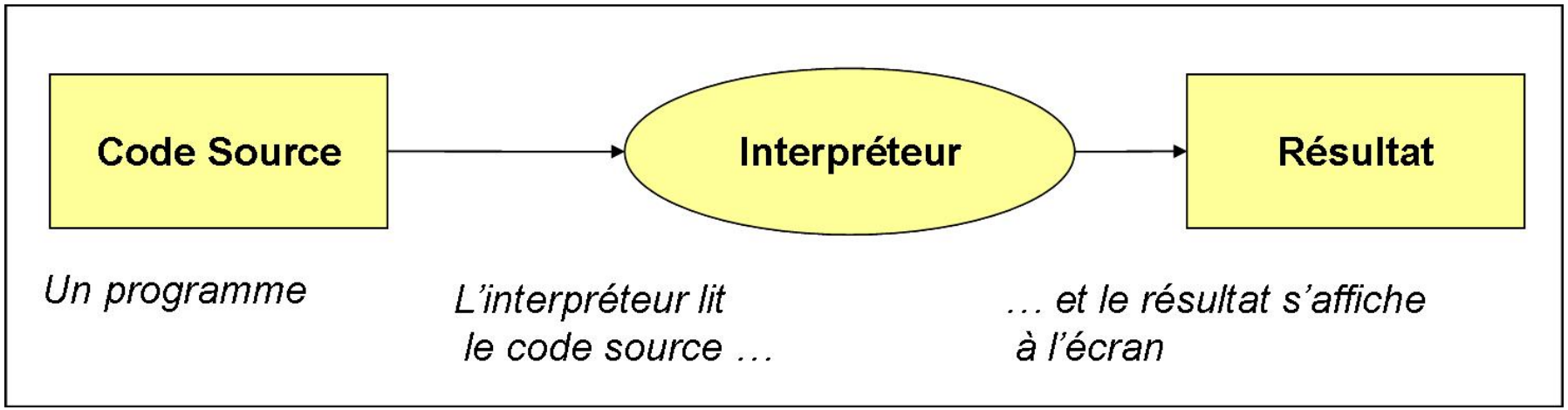
Langage combinant plusieurs paradigmes de programmation.

Exemples

C++, JAVA, PYTHON,PERL,RUBY, ...

Production de programmes : Interprétation

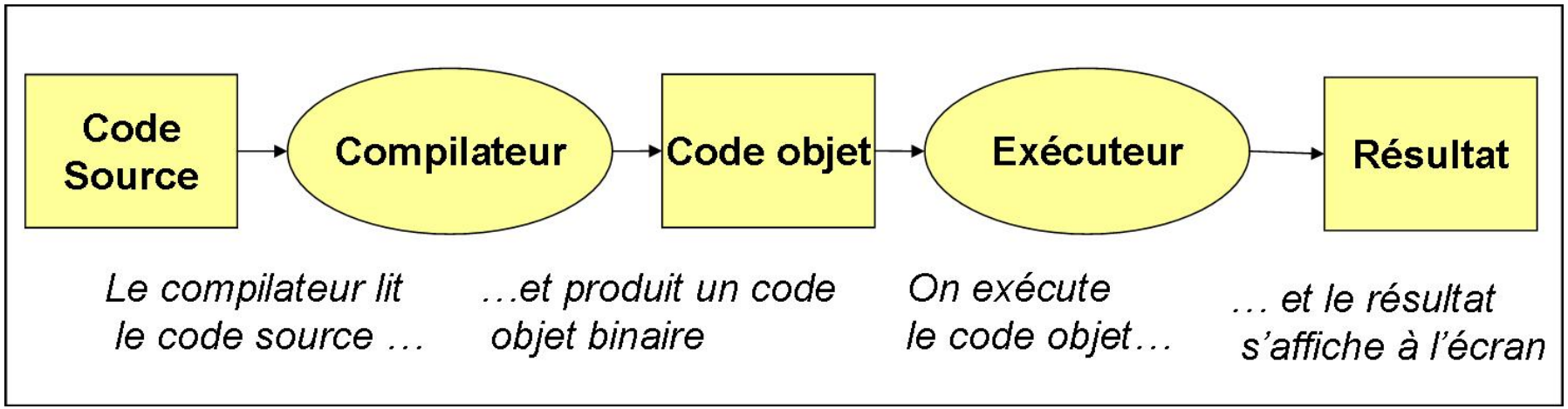
Interprétation



Chaque ligne du code source est traduite au fur et à mesure en instructions qui sont directement exécutées, i.e. l'interpréteur est utilisé à chaque exécution.

Production de programmes : Compilation

Compilation



Compilation = traduction du code ecrit dans un langage dit source en langage objet ou cible (analyse lexicale, syntaxique, sémantique et production du code objet). L'édition de liens ..

Sommaire

Algoritmes récursifs

Définition

Un algorithme est dit récursif s'il s'appelle lui même.

Récusivité

Principe qui consiste à décrire les étapes nécéssaires à la résolution de problèmes en utilisant la résolution du même problème sur des entrées plus petites.

Algoritmes récursifs : exemple

Factorielle d'un nombre

On peut définir la factorielle d'un nombre de la manière suivante :

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n(n-1)! & \text{sinon} \end{cases}$$

Algoritmes récursifs : exemple

Factorielle : algorithme itératif

Algorithm 3 FACTORIEL-ITERATIF (n : entier positif)

1: $x \leftarrow 1$
2: **for** $i \leftarrow 1$ to n **do**
3: $\quad x \leftarrow i * x$
4: **end for**
5: return x

Algoritmes récursifs : exemple

Factorielle :algorithme récursif

Algorithm 4 FACTORIEL-RECURSIF (n : entier positif)

1: **if** $n = 1$ **then**
2: return 1
3: **else**
4: return n FACTORIEL-RECURSIF($n - 1$)
5: **end if**

Règles de conception

Première règle

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne comporte pas d'appel récursif : éviter les calculs infinis.

Cas de base

Les cas de base sont les cas non récursifs d'un algorithme récursif.

Conditions de terminaisons

Les conditions de terminaisons sont les conditions que doivent satisfaire les données dans les cas de base.

Règles de conception

Deuxième règle

Vérifier que tous les appels récursifs effectués terminent bien sur une condition de terminaison.

Théorème

Il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls.

Sommaire

Le langage Python

C'est

- Un langage de scripts (de petits programmes très simples chargés d'une mission très précise sur votre ordinateur) ;
- Un langage impératif interprété (c'est-à-dire que les instructions que vous lui envoyez sont "transcrites" en langage machine au fur et à mesure de leur lecture) contrairement à un langage compilé (C/C++)

Le langage Python

Python est un langage de programmation

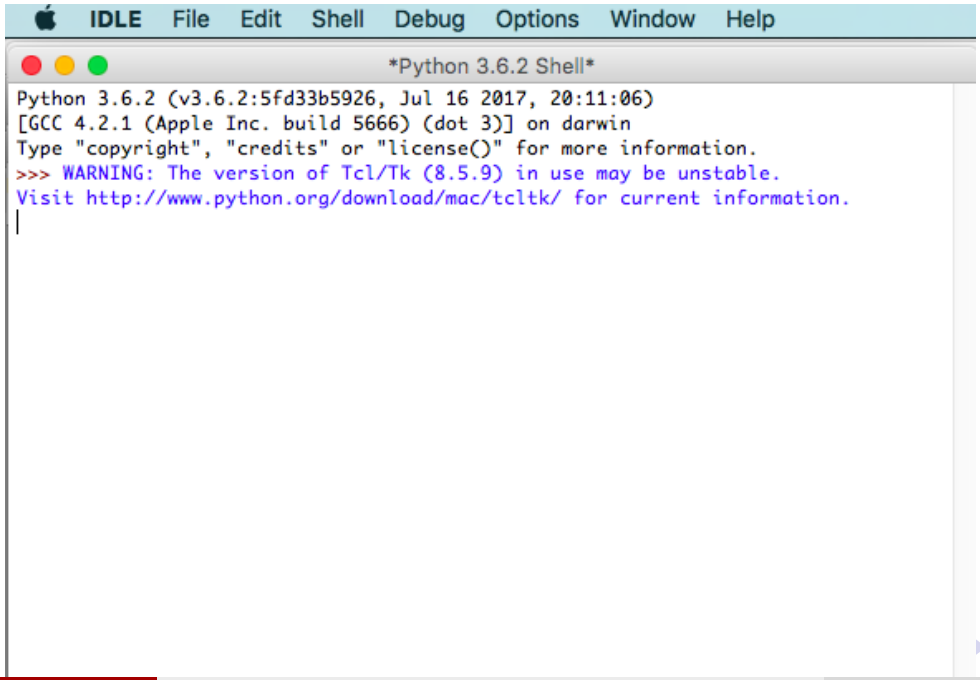
Ce qui définit un langage de programmation

- La façon de représenter symboliquement les structures de données.
- La façon de gérer le contrôle des programmes (que faire et dans quel ordre)

Comme tout langage de programmation, Python fournit un certain nombre de types de données prédéfinis (booléens, entiers, flottants, chaîne de caractères, listes) et de structure de contrôle (les boucles for, while et les conditions if_then_else_).

Installation et Prise en main

Pour installer Python sur votre machine personnelle, vous devez télécharger la dernière version du langage à l'adresse `https://www.Python.org/downloads/`. L'installation de Python génère l'installation d'une interface, appelée IDLE (Python GUI). Cette interface vous permet de saisir des instructions en ligne de commande mais également d'exécuter des programmes Python enregistrés dans des fichiers. L'accès à l'interface IDLE se fait à partir du répertoire où Python a été installé. Il suffit alors de cliquer sur IDLE qui va vous ouvrir l'interface graphique où vous pourrez taper vos instructions Python en ligne de commandes :



Installation et Prise en main

Pour écrire un programme dans un fichier, dans le menu *File*, sélectionnez *New File*. Une nouvelle fenêtre s'ouvre. Tapez votre programme Python dans cette fenêtre (attention aux indentations). Pour exécuter votre programme, allez dans le menu Run et faites Run Modules (ou F5). Il va vous être demandé de faire une sauvegarde de votre fichier (qui a généralement l'extension *.py*), puis votre programme s'exécutera (dans la fenêtre en ligne de commande précédemment ouverte). Le début de l'execution de votre programme est indiqué dans la fenêtre en ligne de commande par le mot clé `RESTART`.

Installation et Prise en main

D'autres outils intégrant un éditeur de texte peuvent aussi être utilisés pour programmer en Python. Exemples : Canopy `https://www.enthought.com/products/canopy/` et IEP (Interactive Editor for Python) `http://www.pyzo.org/iep.html`.

Les structures de données

Structure de données

Une structure de données est définie par 3 choses :

1. Un **type** qui est simplement un nom permettant de classifier les données relevant de ce nom.
2. Un **ensemble** qui définit avec précision les éléments appartenant au type.
3. des **opérations** utilisées par les programmes pour calculer sur ces données

On parle de structures algébriques.

Les booléens

Ce type de données est prédéfini dans Python par :

- son nom bool
- ses valeurs {True,False}
- ses opérations
  - not : $bool \rightarrow bool$,
  - and : $bool \times bool \rightarrow bool$, et
  - or : $bool \times bool \rightarrow bool$

Nom vient de George Boole [1815-1864] mathématicien anglais intéressé par les propriétés algébriques des valeurs de vérité.

Les entiers relatifs

Ce type de données est prédéfini dans Python par :

- son nom int
- son ensemble de valeurs est théoriquement $\mathbb{Z}$ mais est en fait borné selon la machine utilisée.
- ses opérations
  - +, -, *, /, %, ** : $int \times int \rightarrow int$,
  - ==, >, <, >=, <=, != : $int \times int \rightarrow bool$

Les opérations de calcul (+,-,** ...) sont prioritaires sur les opérateurs de comparaison (==,<=, ...).

Quelques exemples d'utilisation des entiers

```
>>> type(46)  #type : fonction pr\'{e}d\'{e}finie fournissant
type 'int'              #le type d'une expression
>>> print 5 * 7   #print : fonction pr\'{e}d\'{e}finie d'impression
35
>>> 5 / 2   # / quotient de la division euclidienne
2
>>> 5 % 2 #  % reste de la division euclidienne
1
>>> 5 ** 3 # fonction d'exponentation
125
>>> 5 < 3 # pr\'{e}dicat de comparaison "strictement inf\'{e}rieur"
False
>>> 5 ==  3 + 2 # op\'{e}ration + est prioritaire sur ==
True
>>> 5 = 3 + 2
SyntaxError: can't assign to literal
```

Les nombres réels

En fait, les nombres réels ne sont pas représentables en machine. On les approxime par des nombres rationnels (i.e. avec des virgules).
Ce type de données est prédéfini dans Python par :

- son nom float
- son ensemble de valeurs est théoriquement $\mathbb{R}$ mais est en fait un sous-ensemble de $\mathbb{Q}$. De plus, il est borné selon la machine utilisée.
- ses opérations
  - +, -, *, /, ** : $float \times float \rightarrow float$,
  - int : $float \rightarrow int$, et float : $int \rightarrow float$
  - ==, >, <, >=, <=, ! = : $float \times float \rightarrow bool$

Les flottants s'écrivent en utilisant le point pour séparer la partie décimale de la partie entière.

Quelques exemples d'utilisation des nombres réels

```
>>> type(46.8)
<type 'float'>
>>> 5.0 / 2.  # le point distingue un flottant d'un entier,
2.5
>>> 5. / 2  # 2 est converti en flottant
2.5
>>> 7.5 * 2
15.0
>>> int (7.5 * 2)
15
>>> int (7.75)
7
>>> float (3)
3.0
```

Les chaînes de caractères

Type immuable (on ne peut pas changer les valeurs)

Description formelle

Une chaîne de caractères est une application $s : S \rightarrow char$ où S est une séquence finie d'entiers positifs $\{0, \ldots, n\}$ et char est l'ensemble des 256 caractères ascii.

Ce type de données est prédéfini dans Python par :

- son nom str
- son ensemble de valeurs est tout mot fini (i.e. une suite de caractères ascii).
- ses opérations
  - $+ : str \times str \rightarrow str$ (concaténation de deux chaînes de caractères),
  - $* : str \times int \rightarrow str$ l'entier doit être positif,
  - len : $str \rightarrow int$ (longueur d'une chaîne de caractères),
  - $==, >, <, >=, <=, != : str \times str \rightarrow bool$

Les chaînes de caractères - suite

En python, une chaîne se décrit en extension en écrivant le mot avec des guillemets : $s =$ "$a_1 \dots a_n$".

La notation $s[i]$ pour $i = 0, \dots, (n-1)$ dénote la lettre à la position $i+1$ dans la chaîne s.

On peut accéder à des sous chaînes de caractères d'une chaîne donnée. Soit s une chaîne de longueur n. s[i :j] avec $i < j$ et $i, j \leq n$, dénote la sous-chaîne $s[i]s[i+1] \dots s[j-1]$

Accès aux sous-chaînes par les (intervalles d')index

```
>>> test ='1234567'
>>> test[3:6]
'456'
>>> test[0]
'1'
>>> test[3]
'4'
>>> test[6]
'7'
>>> test[3:4]
'4'
```

Un programme python

Un programme est une suite de définition de fonction.

Fonction python

Une fonction python se déclare de la façon suivante

```
def nom_fonction (arg1,...,argn):
        # texte de description de la fonction
        Corps de la fonction
```

Le corps de la fonction se défini inductivement à partir :

- des affectations de variables $x = expr$ (x est du type de l'expression expr)
- des conditionnelles
- des boucles for et while

Les résultats des fonctions sont retournés à l'environnement appelant par le mot clé return. Une fonction sans return est appelée une procédure.

Les expressions conditionnelles

Condition en python

Une expression conditionnelle python se déclare de la façon suivante

```
if expression_conditionnelle :
    Instructions
else :
    Instructions
```

Les expressions conditionnelles - suite

Dans le cas où l'on ne souhaite pas rien faire lorsque la condition est fausse, on peut omettre la partie "else".

Enfin, il arrive que l'on ait des "cascades" d'expressions conditionnelles. On peut alors utiliser le mot clé elif.

```
if expression_conditionnelle :
   Instructions
elif expression_conditionnelle :
   Instructions
else :
   Instructions
```

Les boucles While et for

Boucle while

Une boucle while python se déclare de la façon suivante

```
while expression_conditionnelle :
    Instructions
```

Boucle for

Une boucle for python se déclare de la façon suivante

```
for variable in collection :
    Instructions
```

Les collections dans les boucles "for" peuvent être de plusieurs formes :

- "range(ind_début,ind_fin,pas)". La variable va prendre pour valeur à chaque étape de la boucle "for" ind_début,ind_début+pas,...,ind_fin.
- une chaine de caractères ou une liste. Dans ce cas là, la variable va parcourir toute les valeurs se trouvant aux positions allant de 0 à $len(s) - 1$ où s est la chaîne de caractères ou la liste.

Variables locales et variables globales

Lorsqu'une fonction utilise une variable, elle est toujours "locale" à cette fonction.

```
def triple (n) :
    resultat = n * 3
    return resultat
>>> triple(4)
12
>>> resultat            #variable locale n'est pas connue
Traceback (most recent call last): ...
>>> resultat =128 #variable globale
>>> resultat(2)
Traceback (most recent call last): ....
>>> triple(2)
6
>>> resultat #variables globale inchang\'{e}e m\^{e}me si
                        #utilis\'{e}e entretemps comme variable local
128
```

Les listes/tableaux

Description formelle

Une liste est une application $s : S \to TYPE$ où S est une séquences d'entiers positifs $\{0, \ldots, n\}$ et TYPE est l'union de tous les types de données.

Ce type de données est prédéfini dans Python par :

- son nom lists
- son ensemble de valeurs est tout mot fini sur l'ensemble TYPE pris comme alphabet.
- ses opérations
  - $+$: $list \times list \to list$ (concaténation de deux listes),
  - $*$: $list \times int \to list$ l'entier doit être positif,
  - len : $list \to int$ (longueur d'une liste),
  - $==, >, <, >=, <=, != : list \times list \to bool$

En python, une liste se décrit en extension en écrivant le mot entre crochets : $l = [a_1, \ldots, a_n]$. La notation [] dénote la liste vide.

Les listes - suite

En python, les listes sont mutables, i.e., on peut changer les valeurs au différentes positions. On utilise la notation : $l[i] = e$ met l'élément e à la position i dans la liste l

On peut accéder à des sous listes d'une liste donnée. Soit l une liste de longueur n. l[i :j] avec $i < j$ et $i, j \leq n$, dénote la sous-liste $l[i]l[i+1] \ldots l[j]$

Convention : $l[: i]$ et $l[i :]$ dénotent les i premiers et les $n - i$ derniers éléments de la liste l, respectivement.

Quelques illustrations sur les listes

```
>>> type([1,2])
<type 'list'>
>>> [2+3,"toto",2.5]
[5, 'toto', 2.5]
>>> type([2+3,"toto",2.5])
<type 'list'>
>>> [5,[2,2],"toto"]
[5, [2, 2], 'toto']
>>> ["toto",5,"tutu",5.5][1]
5
>>> ["toto",5,"tutu",5.5][1:3]
[5, 'tutu']
>>> "tutu" in ["toto",5,"tutu",5.5]
True
>>> "tu" in ["toto",5,"tutu",5.5]
False
```

Un exemple de programme

Recherche d'un élément dans une liste

```
def find (e,l):
    if l == []:
        return False
    else:
        if e == l[0]:
            return True
        else:
            return find(e,l[1:len(l)])
```