0-M02 – Introduction à la programmation

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

2013 – 2014

Plan du cours

Site web

Slides des cours, versions électroniques des polys, TD, TP...
`http://www.lipn.fr/~coti/cours`

Introduction

Étymologie du mot "informatique"

Formé de la contraction des mots **information** et **automatique**.

- L'informatique est un outil de traitement automatique de l'information.
- On doit alors **définir** comment des informations vont être traitées (automatiquement) par l'ordinateur.

La science informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes – Edsger Dijkstra

- L'ordinateur est un **outil** qui traite l'information comme le programme lui dit de la traiter.

Pourquoi l'algorithmique

Qu'est-ce qu'un algorithme ?

- Un algorithme définit ce que fait un programme
- Il définit quel comportement suivre selon la situation rencontrée

Algorithme : définition

Série d'instructions qui doit être exécutée par un programme.

Définition de Wikipedia :

- Processus systématique de résolution d'un problème permettant de décrire les étapes vers le résultat;
- Suite finie et non-ambiguë d'instructions permettant de donner la réponse à un problème.

Comment décrire un algorithme

Définition de Wikipedia :

- L'**algorithmique** est l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception d'algorithmes.

Algorithmique

Formalisme permettant de décrire la série d'instructions exécutée par un programme indépendamment d'un langage de programmation en particulier.

Les algorithmes sont décrits en **pseudo-code**, compréhensible par le lecteur humain mais assez précis pour transcrire les structures et les instructions de l'algorithme.

Par quoi est constitué un algorithme

Un algorithme permet d'obtenir un résultat à partir de données d'entrée. Il est donc constitué des éléments suivants :

- Un début et une fin ;
- Un nom ;
- Des données d'entrée ;
- Des données de sortie, qui sont le résultat du calcul effectué par l'algorithme ;
- Un ensemble d'instructions exécutées par l'algorithme.

Exemple : algorithme de calcul d'une valeur absolue

```
1 début fonction abs( i: Entier ): Entier
2 |   si i > 0 alors
3 |   |   r ⟵ i
4 |   sinon
5 |   |   r ⟵ −i
6 |   fin si
7 |   retourner r
8 fin fonction
```

Importance d'un bon algorithme

Compréhension et modélisation du problème

- L'algorithme modélise le comportement du programme

Correction du résultat

- Algorithme faux $\rightarrow$ résultat du calcul faux

Efficacité du calcul

- Coût d'un calcul = nombre d'opérations et quantité d'information manipulée
- Ces quantités sont définies par l'algorithme
- Algorithme efficace $\rightarrow$ calcul efficace (et inversement)

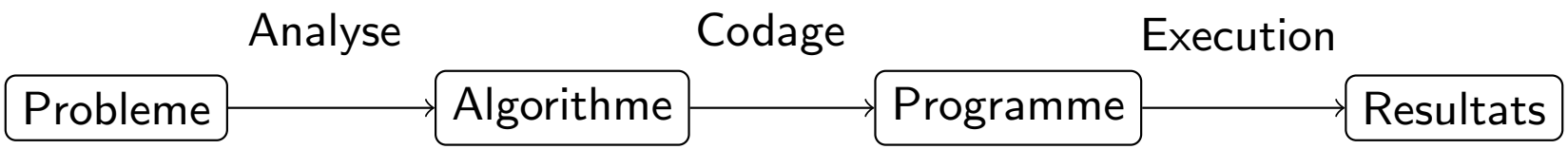
L'étude algorithmique d'un problème est indispensable

La bonne conception d'un algorithme est fondamentale et absolument nécessaire préalablement à l'écriture d'un programme informatique.

Généricité des algorithmes

Les algorithmes sont écrits dans un langage de descriptions des algorithmes (pseudo-code)

- Indépendant du langage de programmation
  $\rightarrow$ Un algorithme peut être implémenté dans n'importe quel langage



Représentation des données

Définition

Une **variable** correspond à l'emplacement mémoire d'une donnée. Elle sert à stocker une valeur d'un certain **type** à un instant donné de l'exécution du programme.

Les variables d'un algorithmes sont :

- D'entrée : données d'entrée de l'algorithme
- De sortie : résultat du calcul effectué par l'algorithme
- Interne à l'algorithme : ni d'entrée ni de sortie mais utilisée à l'intérieur de l'algorithme

Type de variables

Définition

Le **Type** d'une variable est le type de donnée qui pourra être contenu dans cette variable.

Exemples :

- Entier : tout nombre entier
- Booléen : vrai ou faux
- Caractère
- Nombre réel
- Chaîne de caractères
- Tableau...

On ne peut pas mettre une donnée d'un type dans une variable d'un autre type

- Exception : un transtypage est parfois possible (entier dans réel...)

Affectation

Affectation d'une valeur à une variable

Pour écrire une valeur dans une variable, on dit que l'on **affecte** cette valeur à la variable. On le note de la façon suivante :

$$variable \longleftarrow valeur$$

```
1 début
2 |   maVariable : Entier
3 |   maVariable ⟵ 42
4 fin
```

- On **déclare** le type de la variable avant de l'utiliser
- Les déclarations sont généralement rassemblées au début de l'algorithme

Les tableaux

Tableaux

Un tableau est un type particulier de variable. Il contient un **ensemble de variables** de même type, stockées de façon contiguë en mémoire.

Exemple : tableau d'entiers

3	5	2	1	12	5	2	9

Caractéristiques d'un tableau

- Un tableau a une taille fixe
- Il contient un certain type de données

Déclaration d'un tableau

- On le déclare avec le type de données qu'il contient et sa taille

```
1 début
2 |   monTableau[10] :
  |   Tableau d'Entiers
3 fin
```

Les tableaux (suite)

Tableau à plusieurs dimensions

- On donne la taille dans chaque dimension
  - Exemple en 2D (ordre C) : d'abord le nombre de lignes, puis le nombre de colonnes

Accès aux données d'un tableau

- Les cases du tableau sont numérotées de 0 à $N-1$ (si N est la taille du tableau)
- On accède aux données d'un tableau en utilisant l'indice dans ce tableau

```
1 début
2 |   /* Variables d'entree */
3 |   maMatrice[10][10] : Tableau d'Entiers
4 |   /* Variables de sortie */
5 |   i : entier
6 |   /* Affectation */
7 |   i ⟵ maMatrice[2][3]
8 fin
```

Booléen

Booléen

Un booléen est une variable à deux états : **Vrai** ou **Faux**. On représente aussi parfois ces deux états par 1 et 0.

Les opérateurs de comparaison renvoient un booléen. Exemple :

- $1 > 0$ renvoie Vrai
- $1 == 0$ renvoie Faux

Lorsqu'une condition est évaluée, on regarde sa valeur (booléen). Exemple :

```
1 début
2 |   maVar : Entier
3 |   maVar ⟵ 0
4 |   si maVar < 5 alors
5 |   |   maVar ⟵ maVar + 1
6 |   fin si
7 fin
```

- $maVar$ est initialisé à 0
- On teste $maVar < 5$
  - Équivalent à tester $0 < 5$
- La condition vaut **Vrai**
- Donc on exécute le bloc d'instruction après le mot-clé **alors**

Un peu d'algèbre de Boole

On peut évaluer des expressions booléennes en utilisant des opérateurs :

Opérateurs logiques

- ET logique : .
- OU logique : $+$
- OU exclusif : $\oplus$
- Négation : $^-$ ou !

Exemples :

- $a.b$
- $a + b$
- $a \oplus b$
- $\overline{a + b}$ =!$(a + b)$
- $a + \overline{b} = a$+!b

Tables de vérité :

.	0	1
0	0	0
1	0	1

+	0	1
0	0	1
1	1	1

$\oplus$	0	1
0	0	1
1	1	0

La **négation** transforme un Vrai en Faux et inversement :

- $\overline{Vrai} = Faux$
- $\overline{Faux} = Vrai$

Un peu d'algèbre de Boole (suite)

Prenons $a = Vrai$ et $b = Faux$.
Exemples d'expressions booléennes :

- $a + b = Vrai + Faux = Vrai$
- $a + \overline{b} = Vrai + \overline{Faux} = Vrai + Vrai = Vrai$
- $\overline{a.\overline{b}} = \overline{Vrai.\overline{Faux}} = \overline{Vrai.Vrai} = \overline{Vrai} = Faux$

Attention aux parenthèses !

- $(a + b).(c + d)$

Exemples de compositions d'expressions booléennes sur des variables en algorithmique :

- $(a > b) ET (a > 0)$
  - Si $a = 1$ et $b = 2$: $(a > b) = Faux$ donc l'expression vaut $Faux$
  - Si $a = 3$ et $b = 2$: $(a > b) = Vrai$ et $(a > 0) = Vrai$ donc l'expression vaut $Vrai$
- $(a > b) OU (a > 0)$
  - Si $a = 1$ et $b = 2$: $(a > b) = Faux$ et $(a > 0) = Vrai$ donc l'expression vaut $Vrai$
  - Si $a = 3$ et $b = 2$: $(a > b) = Vrai$ et $(a > 0) = Vrai$ donc l'expression vaut $Vrai$

Les tests

Un test est une **structure conditionnelle** : les instructions exécutées dépendent de la réalisation ou non d'une condition.

Définition

Un **test** définit une condition et un comportement à suivre si elle est réalisée. Optionnellement, il peut définir un comportement à suivre dans le cas contraire.

Syntaxe :

- La **condition** est donnée entre les mot-clés **si** et **alors**
- L'**action réalisée si la condition est vérifiée** est donnée après le mot-clé **alors**
- Si on donne une **action à réaliser si la condition n'est pas vérifiée**, elle est introduite par le mot-clé **sinon**
- Le test est terminé par le mot-clé **finsi**

```
1 si condition alors
2 |   action1
3 sinon
4 |   action2
5 fin si
```

Condition d'un test

La condition d'un test est une **expression booléenne**

- Valeurs possibles : VRAI ou FAUX

Elle est évaluée pour décider quel bloc d'instructions exécuter.

On peut tester :

- l'égalité entre deux variables : $var1 == var2$
- la non-égalité entre deux variables : $var1 != var2$
- une relation d'ordre entre deux variables : $var1 > var2$
- ou toute expression renvoyant $Vrai$ ou $Faux$

```
1 début
2 |   maVar : Entier
3 |   maVar ⟵ 0
4 |   si maVar < 5 alors
5 |   |   maVar ⟵ maVar + 1
6 |   fin si
7 fin
```

On peut combiner des expressions booléennes en utilisant les opérateurs logiques ET et OU (attention aux parenthèses) :
$((var1 == var2)\ ET\ (var1 > 0))\ OU\ (var2 < 0)$.

Blocs d'instructions

Un algorithme est structuré par **blocs d'instructions** contenant plusieurs instructions à exécuter séquentiellement.

- Exemple : les instructions à exécuter si la condition d'un test est réalisée
  - Les lignes 5 et 6 sont un bloc
  - La ligne 8 est un bloc
- Des blocs peuvent être **imbriqués**
  - Un bloc d'instructions peut être inclus dans un autre bloc.
  - Les lignes 2 à 9 sont un bloc
  - Les deux blocs dans la condition sont des blocs imbriqués dans ce bloc

```
1  début
2      maVar : Entier
3      maVar ⟵ 0
4      si maVar < 5 alors
5          maVar ⟵ maVar + 1
6          afficher( mavar )
7      sinon
8          maVar ⟵ 0
9      fin si
10 fin
```

- Des instructions d'un bloc sont décalées vers la droite au même niveau
- Un bloc est indiqué par une ligne verticale sur la gauche

Tests imbriqués

On peut **imbriquer** des tests, c'est-à-dire qu'un test peut être effectué dans le corps d'un autre test.

- On effectue un test ligne 4
- Si la condition est réalisée, on exécute le bloc situé entre les lignes 5 et 8
  - On effectue alors un autre test ligne 6
  - Si la condition est réalisée, on exécute le bloc ligne 7
- Sinon, on exécute le bloc ligne 10.

Le test situé entre les lignes 6 et 8 est imbriqué dans le test situé entre les lignes 4 et 11.

```
1  début
2      maVar : Entier
3      maVar ⟵ 0
4      si maVar < 5 alors
5          maVar ⟵ maVar + 1
6          si maVar > 2 alors
7              afficher( mavar )
8          fin si
9      sinon
10         maVar ⟵ 0
11     fin si
12 fin
```

Les boucles

Condition d'arrêt

La condition d'arrêt d'une boucle est une condition (une expression booléenne) qui détermine le moment où une boucle doit arrêter d'exécuter le bloc d'instructions.

Une boucle sert à **répéter** un bloc d'instructions tant qu'une condition de continuation est satisfaite ou que la condition d'arrêt n'est pas satisfaite.
Exemples d'utilisation :

- Parcours d'un tableau, calcul itératif...

* Une boucle utilise un bloc d'instructions : c'est tout le bloc d'instructions correspondant qui est répété.

Il est possible que le bloc d'instructions ne soit pas exécuté du tout (si la condition d'arrêt est déjà satisfaite) ou un nombre infini de fois (souvent un bug).

Boucle **Pour**

On définit un compteur et :

- Une initialisation de ce compteur
  - $i \leftarrow 0$
- Une condition d'arrêt pour sortir de la boucle
  - **à** 9
- Un pas qui modifie le compteur **à la fin** de chaque itération
  - **pas** 1

On termine le bloc avec **finpour**

Exemple : algorithme de remplissage d'un tableau de 10 cases.

```
1 début
2 |   tab[10] : Tableau d'entiers
3 |   pour i ⟵ 0 a 9 pas 1 faire
4 |   |   tab[i] = 2 * i
5 |   fin pour
6 fin
```

Boucle **Pour** (suite)

Le **pas** est n'importe quel modificateur sur le compteur : il peut être négatif, non linéaire...

```
1 début
2 |   pour i ⟵ 10 a 0 pas −2 faire
3 |   |   afficher( i )
4 |   fin pour
5 fin
```

Affichage par le programme :

10
8
6
4
2
0

```
1 début
2 |   pour i ⟵ 1 a 35 pas ∗2 faire
3 |   |   afficher( i )
4 |   fin pour
5 fin
```

Affichage par le programme :

1
2
4
8
16
32

La boucle s'exécute tant que i est inférieur à 35 : on s'arrête à 32.

Boucle Tant que... faire

La boucle *Tant que* exécute un bloc d'instructions tant qu'une condition est vraie : c'est la **condition de boucle**.

- La condition de boucle est définie après le mot-clé **Tant que**
- L'action à effectuer est donnée entre les mot-clés **faire** et **fintq** : on définit un bloc d'instructions qui est répété

Attention aux boucles infinies !

- Il faut que la condition de boucle finisse par être invalidée...

Exemple : algorithme de calcul des puissances de 2 inférieures à 50.

```
1 début
2 |   puissance : Entier
3 |   puissance ⟵ 1
4 |   tant que puissance < 50 faire
5 |   |   afficher( puissance )
6 |   |   puissance ⟵ puissance * 2
7 |   fin tq
8 fin
```

Affichage :

```
1
2
4
8
16
32
```

Boucle **Tant que... faire** (suite)

```
1 début
2 |   puissance : Entier
3 |   puissance ← 1
4 |   tant que puissance < 50 faire
5 |   |   afficher( puissance )
6 |   |   puissance ← puissance * 2
7 |   fin tq
8 fin
```

Détail de l'exécution :

- $puissance$ vaut 1
- $puissance$ est-il inférieur à 50 ? oui donc on exécute le bloc
- Affichage : 1
- $puissance$ vaut 2
- Retour à la ligne 4 : $puissance$ est-il inférieur à 50 ? oui donc on exécute le bloc
- Affichage : 2
- $puissance$ vaut 4
- ...
- Lorsque $puissance$ prend la valeur 64 : la condition ligne 4 n'est plus satisfaite et on sort de la boucle

Équivalence entre les boucles **for** et **tant que ... faire**

On peut écrire une boucle **faire ... tant que** équivalente à une boucle **for** :

- Le compteur est initialisé avant d'entrer dans la boucle **faire ... tant que**
- La condition de boucle est la même que la condition d'arrêt de la boucle **for**
- Le compteur est modifié à la fin du bloc d'instruction exécuté par la boucle **faire ... tant que**

```
1 début
2 |   pour i ⟵ 0 a 9 pas 1 faire
3 |   |   tab[i] = 2 * i
4 |   fin pour
5 fin
```

```
1 début
2 |   i : entier
3 |   i ⟵ 0
4 |   tant que i < 10 faire
5 |   |   tab[i] = 2 * i
6 |   |   i ⟵ i + 1
7 |   fin tq
8 fin
```

Boucle **Faire ... tant que**

- La boucle **Faire ... tant que** exécute un bloc d'instructions **puis** évalue une condition de boucle
- Le bloc d'instructions est répété si la condition de boucle est satisfaite

```
1 début
2     puissance : Entier
3     puissance ← 1
4     faire
5         afficher( puissance )
6         puissance ← puissance * 2
7     tant que puissance < 20;
8 fin
```

Détail de l'exécution :

- $puissance$ vaut 1
- Affichage : 1
- $puissance$ vaut 2
- $puissance$ est-il inférieur à 20 ? oui donc on ré-exécute le bloc : retour à la ligne 4
- Affichage : 2
- $puissance$ vaut 4
- ...
- Lorsque $puissance$ a pris la valeur 32 : la condition ligne 7 n'est plus satisfaite et on sort de la boucle

Différence entre les boucles **Faire ... tant que** et **Tant que ... faire**

- La boucle **Faire ... tant que commence par évaluer** la condition de boucle
  - **Puis** elle exécute le bloc d'instructions si la condition de boucle est validée
- La boucle **Tant que ... faire** exécute le bloc d'instructions **puis elle évalue** la condition de boucle

Algorithme d'attente à un stop :

```
1 début
2 |   vitesse : Entier
3 |   faire
4 |   |   vitesse ⟵ 0
5 |   tant que voituresArrivent == Vrai;
6 |   vitesse ⟵ 50
7 fin
```

Algorithme d'attente à un feu rouge :

```
1 début
2 |   vitesse : Entier
3 |   tant que couleurFeu == rouge faire
4 |   |   vitesse ⟵ 0
5 |   fin tq
6 |   vitesse ⟵ 50
7 fin
```

Avec un stop on s'arrête, puis on regarde si on peut avancer. À un feu de circulation, on s'arrête si le feu est rouge ; si le feu n'est pas rouge, on avance.

Décomposition d'un problème en sous-problèmes

"... diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre."

Discours de la méthode (1637)
René Descartes

Fonctions

Définition

Une **fonction** définit une action effectuée sur un ensemble de paramètres qui produit un résultat retourné comme valeur de sortie de la fonction.
Elle fournit une **abstraction** pour cette action vis-à-vis des algorithmes qui l'appellent.

Analogie avec les fonctions mathématiques : $f : x \mapsto f(x)$

- On définit la fonction une fois
- On l'appelle autant de fois qu'on veut

Avantages : factorisation de code, abstraction pour la conception de l'algorithme... Permet de se concentrer sur l'algorithme lui-même plutôt que sur les détails.

Paramètres et résultat

On donne les paramètres et leur type

- On déclare les paramètres entre parenthèses
- On les utilise dans la fonction en utilisant **ce nom** : ce sont des variables
- Il existe des paramètres
  - D'entrée : utilisés dans la fonction
  - De sortie : modifiés dans la fonction
  - Ou les deux : utilisés et modifiés dans la fonction

On déclare le type retourné

- Déclaration de la fonction suivie par **: type**
- On sort de la fonction avec le mot clé **retourner** et la valeur retournée

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

Appel de fonction

On appelle la fonction à partir d'un autre algorithme

- On lui passe en paramètres des variables de l'algorithme appelant
- La valeur retournée est mise dans une variable de l'algorithme appelant

```
1 début programme
2 |   nombreDepart : Entier
3 |   calcul : Entier
4 |   calcul ⟵ carre ( nombreDepart )
5 fin programme
```

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

La fonction fournit donc une **abstraction** de l'action qu'elle réalise paramétrée par les variables passées lors de l'appel.

Procédures

Une procédure effectue une **action**. Elle ne retourne rien.

- Affichage : pas de résultat de calcul à récupérer
- Calcul sur des **variables de sortie**

On sort de la procédure

- À la fin du bloc d'instructions principal
- Ou quand on rencontre le mot-clé **Retour**

Procédures (exemple)

```
1 début programme
2 |   puissance : Entier
3 |   puissance ( puissance )
4 |   afficher ( puissance )
5 fin programme
```

```
1 début procédure puissance( petitepuissance: Entier Sortie )
2 |   petitepuissance ⟵ 1
3 |   tant que Vrai faire
4 |   |   petitepuissance ⟵ petitepuissance × 2
5 |   |   if petitepuissance > 200 then
6 |   |   |   retourner
7 |   |   end if
8 |   fin tq
9 fin procédure
```

Visibilité des variables

Les variables déclarées dans la fonction

- ne sont visibles **que** dans la fonction

Les variables déclarées dans le programme appelant

- ne sont visibles **que** dans le programme appelant
- ne sont **pas** visibles dans la fonction

Les variables passées en paramètre

- sont appelées par leur nom dans le programme appelant dans l'appel de la fonction
- sont appelées par le nom utilisé pour les déclarer dans la fonction elle-même

Visibilité des variables

Les variables sont visibles uniquement dans la fonction, la procédure ou le programme dans lequel elles ont été déclarées, et dans aucune des fonctions ou procédures appelées ou qui l'appellent.

Approche descendante

Les fonctions et les procédures fournissent une **abstraction sur les actions réalisées par le programme**

- Possibilité de les utiliser pour se concentrer sur la structure du programme plutôt que sur la résolution des sous-problèmes

On décompose le problème en **sous-problèmes**

- On fait dans un premier temps l'hypothèse qu'ils sont résolus
- On s'en occupe plus tard

La conception de l'algorithme dans sa globalité est ainsi **plus simple**

- Décomposer pour mieux maîtriser

On retarde le plus possible le moment d'effectuer les calculs

Approche descendante : exemple

Théorème de Pythagore : *"le carré de l'hypoténuse d'un triangle rectangle est égal à la somme des carrés des longueurs des deux autres côtés"*.

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2  |   /* variables internes */
3  |   carre1 : Entier
4  |   carre2 : Entier
5  |   hypotenusecarre : Entier
6  |   hypotenuse : Entier
7  |   /* on calcule la somme des carrés des côtés */
8  |   carre1 ⟵ carré( cote1 )
9  |   carre2 ⟵ carré( cote2 )
10 |   hypotenusecarre ⟵ carre1 + carre2
11 |   /* on prend la racine carrée de la somme et on retourne le
   |   résultat */
12 |   hypotenuse ⟵ racine( hypotenusecarre )
13 |   retourner hypotenuse
14 fin fonction
```

Approche descendante : exemple (suite)

```
1 début fonction carré( nombre: Entier ): Entier
2 |   moncarre : Entier
3 |   moncarre ⟵ nombre × nombre
4 |   retourner moncarre
5 fin fonction
```

```
1 début fonction racine( nombre: Entier ): Entier
2 |   maracine : reel
3 |   maracine ⟵ √nombre
4 |   retourner maracine
5 fin fonction
```

Récursivité

Une fonction ou une procédure peut **s'appeler elle-même**

- Elle est alors partiellement définie à partir d'elle-même

```
1 début fonction calcul( var: Entier ): Entier
2 |   si var == 1 alors
3 |   |   retourner var
4 |   sinon
5 |   |   retourner var×calcul( var − 1 )
6 |   fin si
7 fin fonction
```

- Correspond bien aux relations de récurrence
- Attention au point d'arrêt !

Introduction à Python

Python est un **Langage de script**

- Rapidité de développement
- Utilisation pour des scripts d'administration système, analyse de fichiers textuels (logs...)
- Langage pour le web : développement d'applications web, scripts CGI, serveurs...
- Accès aux bases de données relationnelles

Python est un **langage de programmation**

- Programmes complets en Python
- Interfaçage facile avec des bibliothèques dans d'autres langages (C, C++, Fortran...)
- Accès aux interfaces graphiques facilité
- Permet de se concentrer sur l'algorithme plutôt que l'implémentation : calcul scientifique pour les non-informaticiens...

Exécution de programmes Python

Deux moyens d'exécuter des scripts Python :

- En **ligne de commande**: dans l'interpréteur interactif
  - On lance l'interpréteur

```
1 coti@maximum:~$ python
2 Python 2.7.3rc2 (default, Apr 22 2012, 22:30:17)
3 [GCC 4.6.3] on linux2
4 Type "help", "copyright", "credits" or "license" for more
      information.
5 >>> print 3
6 3
```

  - Rapidité de mise en place
  - Permet de tester des choses
- En **exécutant un script**
  - Fichier exécutable
  - Deux possibilités :
    - Exécution directe et le script appelle l'interpréteur
    - Attention aux droits (+x)

```
1 coti@maximum:~$ ./monscript.py
```

    - Lancement dans l'interpréteur

```
1 coti@maximum:~$ python ./monscript.py
```

Scripts Python

Si il est lancé seul, un script Python doit remplir deux conditions :

- Être **exécutable** (+x)
- Spécifier le chemin vers l'interpréteur : c'est le **shebang**

On place le shebang au début du fichier :

```
1 #!/usr/bin/python
```

Attention : si le shebang ne pointe pas vers le bon interpréteur, erreur.

```
1 coti@thorim:/tmp$ cat demo.py
2 #!/usr/python
3 coti@thorim:/tmp$ ./demo.py
4 -bash: ./demo.py: /usr/python: bad interpreter: No
      such file or directory
```

Possibilité de faire coexister plusieurs versions de Python sur le système.

Commentaires en Python

Pour commenter une ligne de code on utilise #

- Une ligne commentée n'est pas exécutée
- Commente tout ce qui suit la ligne
- Commente une et une seule ligne : le commentaire s'arrête à la fin de la ligne

Pour commenter plusieurs lignes, on encadre la section à commenter par ''' (triple quote) ou """ (triple double quote)

- Le commentaire commence au triple quote
- Il se termine au triple quote suivant
- Impossible d'imbriquer des commentaires sur plusieurs lignes

```
1  #!/usr/bin/python
2
3  '''
4  un commentaire
5  sur plusieurs lignes
6  '''
7
8  # un commentaire sur une ligne
```

Affichage d'une chaîne de caractères

On affiche une chaîne de caractères avec l'instruction `print`

- En suivant directement l'instruction `print`

```
1 >>> print "toto"
2 toto
3 >>> a = 2
4 >>> print a
5 2
```

- Ou n'importe quoi entre parenthèses

```
1 >>> print( a )
2 2
```

- Les chaînes de caractères sont données entre guillemets, sinon elles sont interpretées comme des noms de variables

```
1 >>> print "toto"
2 toto
```

Affichage d'une chaîne de caractères

Opérateur de concaténation : +

- Attention, le + est d'abord interprété comme un opérateur mathématique si il est utilisé sur autre chose que des chaînes de caractère

```
1 >>> print a + 3
2 5
3 >>> print a, 3
4 2 3
```

Possibilité d'afficher deux éléments de types différents à la suite avec , (séparés par un espace)
On ne peut concaténer que des variables de même type :

```
1 >>> print "toto" + 3
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: cannot concatenate 'str' and 'int'
      objects
```

Solution : convertir la variable en chaîne de caractères

```
1 >>> print "toto" + str( 3 )
2 toto3
3 >>> print "toto", 3
4 toto 3
```

Type des variables

Type **implicite** :

- On ne déclare pas les variables
- Le type dépend de la première initialisation

Exemple :

```
1  maVar1 = 5          # un entier
2  maVar2 = 3.0        # un reel
3  maChaine = "Toto" # une chaine de caracteres
```

Si on change de type :

- La première variable est détruite
- Une nouvelle est créée sous le même nom

```
1  maVar = 5          # un entier
2  maVar = 3.0        # un reel
```

Types fournis par Python

Entiers :

- integer : 32 bits
- long integer : 64 bits

```
1 >>> a = 3    # entier
2 >>> b = 1L   # entier long
```

Réels :

- type float : 64 bits, virgule flottante

Complexes :

- partie réelle et partie imaginaire
- type float pour chacun des nombres

```
1         >>> z = 9+5J
2         >>> print z.imag
3         5.0
4         >>> print z.real
5         9.0
6         >>> print z
7         (9+5j)
```

Chaînes de caractères

Chaînes de caractères :

- données entre ' (simple quote) ou entre " (double quote)
- si la chaîne contient un ' ou un " : on l'échappe avec un \ pour qu'il ne soit pas interprété

```
1  str = "Vive la prog"
2  str2 = 'Python c\'est bon'
3  str3 = "Pruch\'ella duri"
```

Concaténation avec + :

```
1  >>> toto = "blabla"
2  >>> titi = "blublu"
3  >>> tata = toto + titi
4  >>> print tata
5  blablablublu
```

Introspection

Définition : *Connaissance qu'une entité a d'elle-même.*

Ici : possibilité qu'a le programme d'examiner la structure d'une variable, de connaitre son type.

- On utilise la fonction `type()`

```
1     >>> i = 5
2     >>> type( i )
3     <type 'int'>
4     >>> j = 6L
5     >>> type( j )
6     <type 'long'>
7     >>> z = 9+5J
8     >>> type( z )
9     <type 'complex'>
10    >>> type( z.real )
11    <type 'float'>
```

Conversion de type

But : obtenir une variable d'un type donné contenant la même valeur qu'une autre variable d'un autre type.

- On utilise une fonction qui **construit** la nouvelle variable du type voulu
- Cette fonction porte le nom de ce type
  - Exemple : pour un entier, fonction `int()`

```
1  >>> a = 7.0
2  >>> type( a )
3  <type 'float'>
4  >>> b = int( a )
5  >>> print b
6  7
7  >>> type( b )
8  <type 'int'>
```

Si la variable résultat ne peut pas contenir toute l'information de la variable initiale, la valeur est tronquée

- Exemple : un réel converti en entier $\rightarrow$ l'entier contient la partie entière du réel

```
1  >>> a = 7.7
2  >>> b = int( a )
3  >>> print b
4  7
```

Saisie d'une valeur par l'utilisateur

Saisie d'une chaîne de caractères par l'utilisateur : `raw_input()`

- Ce qui est lu est *toujours* considéré comme une chaîne de caractères
- Paramètre (optionnel) : invite affichée à l'écran

```
1  >>> a = raw_input( "entrer une valeur " ))
2  entrer une valeur toto
3  >>> type( a )
4  <type 'str'>
5  >>> b = raw_input()
6  3
7  >>> type( b )
8  <type 'str'>
```

Saisie d'une valeur par l'utilisateur : `input()`

- Le type de la variable obtenue dépend de la valeur saisie

```
1  >>> c = input( "---> " )
2  ---> 4
3  >>> type( c )
4  <type 'int'>
5  >>> d = input( )
6  toto
7  Traceback (most recent call last):
8    File "<stdin>", line 1, in <module>
9    File "<string>", line 1, in <module>
10 NameError: name 'toto' is not defined
11 >>> d = input( )
12 "toto"
```

Blocs en Python

L'indentation est primordiale

- C'est le niveau d'indentation qui définit les blocs
- Utilisation de la **tabulation**

Les blocs n'ont pas de délimiteur de fin explicite : c'est le retour au niveau d'indentation inférieur (vers la gauche) qui l'indique.

```
1  if True:
2          # debut d'un bloc
3          print "on est dans le bloc"
4  # fin d'un bloc
5  print "retour dans le bloc d'origine"
```

Chaque bloc imbriqué est situé à un niveau d'indentation supérieur : une tabulation supplémentaire vers la droite.

Tests

Mot-clé `if` suivi de deux points `:`

- Le bloc à exécuter est indenté vers la droite

```
1  a = 5
2  if a > 0:
3       print "a est positif"
```

Alternative : mot-clé `else` suivi de deux points `:`

```
1  a = 5
2  if a >= 0:
3       print "a est positif ou nul"
4  else:
5       print "a est negatif"
```

Possibilité d'enchainer les tests avec `elif`

```
1  a = 5
2  if a > 0:
3      print "a est positif"
4  elif a == 0:
5      print "a est nul"
6  else:
7       print "a est negatif"
```

Boucle for

On peut répéter un ensemble d'instructions avec la boucle **for**

- Correspond à la boucle **Pour** vue en algorithmique
- On itère sur un **ensemble de valeurs**

Par exemple, l'ensemble d'entiers compris entre deux valeurs : fonction `range()`

- Un seul argument : entiers compris entre 0 et cet argument (exclu)

```
1 >>> print range( 5 )
2 [0, 1, 2, 3, 4]
```

- Deux arguments : entiers compris entre le 1er et le 2eme argument

```
1 >>> print range( 2, 5 )
2 [2, 3, 4]
```

- Trois arguments : entiers compris entre le 1er et le 2eme argument avec un pas correspondant au 3eme argument

```
1 >>> print range( 1, 10, 3 )
2 [1, 4, 7]
```

Boucle for (suite)

Une utilisation de la boucle `for` consiste donc à intérer sur un ensemble de valeurs d'un compteur

- Ensemble de valeurs suivi de deux points :
- Le bloc à exécuter est indenté d'un cran

```
1 for i in range( 0, 10, 2 ):
2         print i
```

correspond à :

pour $i \longleftarrow 0$ **a** 9 **pas** 2 **faire**
    $affiche r(i)$
fin pour

Affichage obtenu :

```
1 coti@thorim:~$ ./demo.py
2 0
3 2
4 4
5 6
6 8
```

Boucle conditionnelle **while**

Un autre type de boucle est la boucle conditionnelle : un bloc d'instructions est répété tant qu'une condition est réalisée.

- La condition est évaluée **avant** d'exécuter le bloc d'instructions
- Si elle n'est jamais réalisée, on n'entre jamais dans la boucle

On utilise le mot-clé `while`

- Condition suivie de deux points `:`
- Le bloc à exécuter est indenté d'un cran

```
1  i = 0
2  while i < 10 :
3      print i
4      i = i + 1
```

En langage algorithmique, cela correspondrait à une boucle `tant que` :

```
1  i : Entier
2  i ⟵ 0
3  tant que i < 10 faire
4  |   afficher(i)
5  |   i ⟵ i + 1
6  fin tq
```

Fonctions et procédures

La définition d'une fonction est introduite par le mot-clé `def`

- Suivi du nom de la fonction
- Entre parenthèses, ses arguments (parenthèses vides si aucun)
- Enfin, la ligne est terminée par deux points `:`

Le corps de la fonction est un bloc : on l'**indente** donc d'un cran vers la droite

On peut sortir d'une fonction de deux façons :

- En arrivant à la fin de la fonction : retour à l'indentation de niveau maximal (complètement à gauche), dans le cas d'une procédure
- En exécutant le mot-clé `return`
  - Soit seul : fin d'une procédure (ne retournant rien)
  - Soit suivi d'une variable qui est retournée par la fonction

```
1 def maFonction( a ):
2         print "parametre : " + str( a ) + " de type " + str( type( a ) )
3         return type( a )
```

Appel d'une fonction

On appelle une fonction **par son nom**, en lui passant ses **paramètres entre parenthèses**

```
1  def maFonction( a ):
2          print "parametre : " + str( a ) + " de type " + str( type( a ) )
3          return type( a )
4
5  # ailleurs dans le programme
6  maFonction( 5 )
7  maFonction( "toto" )
```

Attention : pas de vérification du type des arguments passés

- Source d'erreurs pas directement détectées : c'est lors de l'utilisation de la variable de type incorrect dans la fonction que l'erreur est annoncée (si elle l'est...)
- Rend possible l'appel de la fonction avec des arguments de différents types

Point d'entrée dans le programme

L'exécution d'un script Python commence par la **première ligne en-dehors de toute fonction**

- On exécute la première ligne du bloc de plus haut niveau qui ne soit pas une définition de fonction

```
1 def maFonction( a ):
2         print "parametre : " + str( a ) + " de type " + str( type( a ) )
3         return type( a )
4
5 maFonction( 5 ) # premiere ligne executee
```

Les programmes Python complexes sont souvent composés de plusieurs *modules* (plus de détails plus tard)

- Nom du module dans lequel on se trouve : variable `__name__`
- Module principal = `__main__`

Généralement, on commence par tester si on se trouve dans le module principal

- Si c'est le cas, on effectue nos appels de fonction
- Intérêt : écrire des modules auto-suffisants ou utilisables par d'autres scripts

```
1 if __name__ == "__main__":
2     appel_fonction()
```

Collections

Définition : une **collection** est une structures de données permettant de stocker des ensembles.

Trois types en Python :

- Les listes
- Les tuples
- Les dictionnaires

Les opérations que l'on peut effectuer sur chacune sont spécifiques au type de collection.

Particularité : les éléments contenus dans les collections ne sont pas forcément tous du même type.

Les listes

Ensemble d'éléments

- non triés
- modifiables

Notation :

- La liste est donnée entre crochets
- Les éléments sont séparés par des virgules

```
1    >>> l = []
2    >>> type( l )
3    <type 'list'>
4    >>> m = [ 5, 7.6, "bonjour"]
```

Une liste est *ordonnée* : l'ordre dans lequel ses éléments sont disposés a de l'importance.

- Pour accéder à un élément de la liste : utilisation de l'opérateur crochets
- La numérotation des indices commence à 0.

```
1    >>> print m[0]
2    5
```

Opérations sur les listes

Nombre d'éléments dans une liste : fonction `len()`

```
>>> len( m )
3
```

Ajouter un élément dans une liste :

- À la fin de la liste : fonction `append()`

```
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> jour.append( 'dimanche' )
>>> print jour
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'dimanche']
```

- À un endroit précis de la liste : fonction `insert()`

```
>>> jour.insert( 5, 'samedi')
>>> print jour
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi',
'dimanche']
```

Compter le nombre de fois où un élément apparait dans le liste : fonction `count()`

```
>>> lst5 = [ 3, 5, 2, 4, 4, 3]
>>> lst5.count( 4 )
2
```

Opérations sur les listes

Retirer un élément d'une liste : fonction `remove()`

```
1 >>> jour.remove( 'samedi' )
2 >>> print jour
3 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'dimanche']
```

Si l'élément se trouve plusieurs fois dans la liste, seul le premier est retiré.

Retirer un élément d'une liste en le retournant : fonction `pop()`

- Si un indice est passé en paramètre : on retire l'élément correspondant à cet indice
- Si pas de paramètre passé : on retire le dernier élément de la liste

```
1 >>> jour.pop()
2 'dimanche'
3 >>> print jour
4 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
5 >>> jour.pop(2)
6 'mercredi'
7 >>> print jour
8 ['lundi', 'mardi', 'jeudi', 'vendredi']
```

Opérations sur les listes

Concaténation de listes : opérateur `+`

```
1 >>> lst1 = [1, 2, 3]
2 >>> lst2 = [9, 8, 7]
3 >>> lst3 = lst1 + lst2
4 >>> print lst3
5 [1, 2, 3, 9, 8, 7]
```

Tri des éléments d'une liste : fonction `sort()`

```
1 >>> lst2.sort()
2 >>> print lst2
3 [7, 8, 9]
```

Assemblage des éléments d'une liste sous forme d'une chaîne de caractères : fonction `join()` (avec une autre chaîne de caractères)

```
1 >>> lst4 = [ 'i', 'u', 'tv' ]
2 >>> "".join(lst4)
3 'iutv'
```

Les tuples

Tuple : structure de donnée proche d'une liste, mais **non modifiable**

- La liste est donnée entre parenthèses
- Les éléments sont séparés par des virgules

Opérations sur les tuples : les mêmes que pour les listes, en-dehors des opérations de modifications.

```
1 >>> jour = ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi')
2 >>> jour.append( 'dimanche' )
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 AttributeError: 'tuple' object has no attribute 'append'
```

Pourquoi utiliser des tuples :

- Implémentés de façon plus simple que les listes : ils utilisent moins de ressources (en particulier en mémoire).
- Pour s'assurer qu'un ensemble ne sera pas modifié par une autre partie du programme

Pourquoi ne pas utiliser des tuples :

- Non modifiables : plus contraignants

Les dictionnaires

Dictionnaire : structure de donnée qui effectue une association entre une **clé** et une **valeur**.

Notation :

- Définition en utilisant des accolades.
- Couples clé-valeur séparés par des virgules
- Une clé est séparée de sa valeur par deurs points **:**

```
1 >>> dic = { 'pomme' : 'fruit' , 'poire' : 'fruit', 'poireau' :
2 'legume' }
```

Opérations sur les dictionnaires

Accéder aux valeurs : on donne la clé entre crochets

```
1 >>> dic['poireau']
2 'legume'
```

Ajouter une nouvelle paire : on affecte la valeur à sa clé passée entre crochets.

```
1 >>> dic['tomate'] = 'fruit'
2 >>> print dic
3 {'poire': 'fruit', 'tomate': 'fruit', 'poireau': 'legume', 'pomme': 'fruit'}
```

Suppression d'un couple :

- Suppression en retournant la valeur : fonction `pop()` en passant la clé de l'élément à supprimer

```
1 >>> dic.pop( 'poireau' )
2 'legume'
3 >>> print dic
4 {'poire': 'fruit', 'tomate': 'fruit', 'pomme': 'fruit'}
```

- Suppression simple : opérateur `del` sur un élément désigné par sa clé entre crochets

```
1 >>> del dic['pomme']
2 >>> print dic
3 {'poire': 'fruit', 'tomate': 'fruit'}
```

Opérations sur les dictionnaires

Liste des clés et des valeurs : fonctions `keys()` et `values()`.

```
1 >>> dic.keys()
2 ['poire', 'tomate', 'poireau', 'pomme']
3 >>> dic.values()
4 ['fruit', 'fruit', 'legume', 'fruit']
```

Remarque : ce sont des listes.

Existence d'une clé dans le dictionnaire : fonction `has_key()`.

```
1 >>> dic.has_key( 'tomate' )
2 True
```

Éléments d'une collection

On peut effectuer une boucle `for` en itérant sur les éléments contenus dans une collection : utilisation du mot-clé `in`

```
1 lst = [1, 4, 6]
2 for i in lst:
3     print i
```

La variable d'itération contient l'élément de la liste sur lequel on se trouve.

On peut tester l'`appartenance` à une collection :

```
1 lst = [1, 4, 6]
2 if 4 in lst:
3     print 4 est present
```

Éléments d'une collection

Sur un dictionnaire :

- Itération sur les clés des couples avec `in` :

```
1  >>> for d in dic:
2  ...     print d
3  ...
4  poire
5  tomate
6  poireau
7  pomme
```

La variable d'itération contient la clé du couple sur lequel on se trouve.

- Itération sur les couples avec la fonction `iteritems()`

```
1  >>> for k, v in dic.iteritems():
2  ...     print k, v
3  ...
4  poire fruit
5  tomate fruit
6  poireau legume
```

On récupère la clé et la valeur.

Utilisation de collections pour représenter des tableaux

On représente généralement les tableaux par des listes

- On accède aux éléments avec l'indice correspondant entre crochets

```
1 >>> tab = [ 1, 3, 6, 2, 4 ]
2 >>> print tab[2]
3 6
```

Pour aller plus loin : modules de calcul scientifique numpy et scipy

- Numpy définit un type `array` et un type `matrix`
- Fournissent des opérations sur ces matrices

Collections imbriquées

On peut **imbriquer** des collections : mettre des collections dans les champs d'une collection

- C'est notamment une façon de représenter un tableau à plusieurs dimensions

```
1  >>> lst = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
2  >>> print lst
3  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
4  >>> print lst[2]
5  [7, 8, 9]
6  >>> print lst[2][1]
7  8
```

Précaution importante

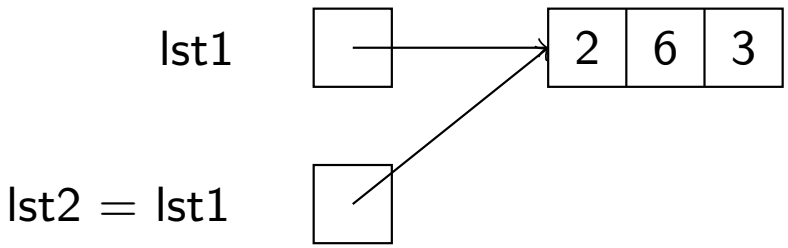
Attention : Python fonctionne en désignant les variables par leur **référence**. Une référence désigne l'adresse de début de l'espace mémoire occupé par la variable.

Exemple :

- Déclaration d'une liste `lst1`
- `lst1` contient en réalité la référence de cette liste
- On copie `lst1` dans une autre variable
- C'est en réalité la référence de la liste qui sera copiée :

```
1 >>> lst1 = [ 2, 6, 3 ]
2 >>> lst2 = lst1
3 >>> print lst2
4 [2, 6, 3]
```

- On parle de **shallow copy**



Précaution importante

Si on effectue une modification sur la liste pointée par `lst2` :

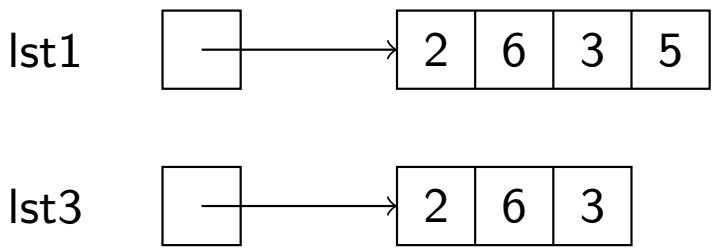
- En mémoire, `lst1` et `lst2` désignent la **même entité**
- La modification est donc effectuée sur la liste pointée par `lst1`

```
1 >>> lst2.append( 5 )
2 >>> print lst1
3 [2, 6, 3, 5]
```

Pour faire une vraie copie : copie explicite de l'élément

- On crée un nouvel élément et on recopie dedans le contenu de l'élément à copier
- On parle de **deep copy**

```
1 >>> lst3 = list( lst1 )
2 >>> print lst3
3 [2, 6, 3, 5]
4 >>> lst3.append( 1 )
5 >>> print lst3, lst1
6 [2, 6, 3, 5, 1] [2, 6, 3,
      5]
```



Qu'est-ce qu'un module

Définition

Un module est un **regroupement de procédures et de fonctions**, dont le but est de les mettre à disposition de plusieurs programmes appelant ces procédures et fonctions.

Intérêt :

- Réutilisation de code : on écrit du code une fois, on peut le réutiliser dans d'autres programmes
- Souplesse de programmation : permet de découper l'architecture du programme selon les tâches à remplir
- Dans le cadre d'un développement collaboratif : chacun développe un module, on appelle des fonctions des modules écrits par les autres

Exemples :

- Fonctions mathématiques : `math` : fournit des fonctions mathématiques telles que `ceil()` et `floor()`, `exp()`, `pow()` et `log()`, `sqrt()`, les fonctions de trigonométrie...
- La tortue utilisée en TP est disponible sous forme d'un module
- Modules de compression : `zlib`, `gzip`, `bz2`...
- Interfaces avec le système d'exploitation : `os`

Que contient un module

Que met-on dans un module ?

- Des **fonctions** et des **procédures**
- Des **constantes**

Par exemple, dans le module `math` on trouve :

- Des fonctions mathématiques : `pow(x, y)` retourne x^y (`x` élevé à la puissance `y`)
- Des constantes mathématiques : π et e

On peut aussi mettre une fonction permettant d'exécuter le module comme un script (fonction `main()`) :

```
1 if __name__ == "__main__":
2     test_du_module()
```

Attention aux constantes : ne pas modifier leur valeur !!

- Par définition, une constante ne doit pas être modifiée
- Python ne fournit pas de moyen de protection
- Règle de programmation : ne pas modifier la valeur d'une constante située dans un module

Écriture d'un module

On écrit simplement des définitions de fonctions et de procédures.

Exemple : un module qui trace des polygones avec la tortue

```
1  #!/usr/bin/python
2
3  from turtle import *
4
5  VERSION = 0.9
6
7  def polygone( x, nb ):
8      for i in range( nb ):
9          forward( x )
10         left( 360/nb )
11
12 def carre( x ):
13     for i in range( 4 ):
14         forward( x )
15         left( 90 )
```

On définit une constante :

- `VERSION`, en majuscules (convention pour les constantes)

Et deux procédures :

- `carre`, qui va tracer un carré
- `polygone`, qui va tracer un polygone régulier

Test de notre module

On peut exécuter ce module comme un script

- Intérêt : tester le module
- On écrit une série d'appels aux fonctions de ce module

Utilisation du nom du module courant :

- Obtenu dans la variable `__name__`
- Toujours à `__main__` quand le script est éxécuté, sinon contient le nom du module

```
1  def main():
2      print "Test du module mod_tortue"
3      carre( 30 )
4      polygone( 30, 6 )
5
6
7  if __name__ == "__main__":
8      main()
```

Chargement d'un module

L'utilisation d'un module se fait en deux temps :

- Chargement du module : mot-clé `import`
- Utilisation du module

Deux façons d'importer un module :

- Importer tout le module :

```
1 import turtle
```

- Importer des parties spécifiques du module :

```
1 from turtle import forward, left
```

NB : avec la seconde méthode d'import, on peut importer l'intégralité du module avec une étoile :

```
1 from turtle import *
```

Utilisation du contenu d'un module

L'utilisation de ses fonctions, procédures et constantes dépend de la façon dont on l'a importé.
Import d'un module en entier :

```
1 import turtle
```

On appelle son contenu en utilisant la notation pointée : nom du module, point, nom de la fonction ou de la constante :

```
1 turtle.forward( 100 )
```

Import de parties spécifiques d'un module :

```
1 from turtle import forward, left
```

On appelle directement les fonctions ou les constantes :

```
1 forward( 100 )
```

Attention avec la seconde méthode, risque de conflit si deux fonctions ou deux constantes portent le même nom.

Informations sur le contenu d'un module

La fonction `dir()` nous donne le contenu d'un module importé

```
1 >>> import time
2 >>> dir( time )
3 ['__doc__', '__file__', '__name__', '__package__', 'accept2dyear', 'altzone
      ', 'asctime', 'clock', 'ctime', 'daylight', 'gmtime', 'localtime', '
      mktime', 'sleep', 'strftime', 'strptime', 'struct_time', 'time', '
      timezone', 'tzname', 'tzset']
```

Un module particulier : `__builtin__`

- Module standard
- Réunit les fonctions fournies par le système par défaut (built-in)

`dir( __builtin__ )` donne la liste des fonctions fournies par le système.

Où sont installés les modules

Lorsqu'un module est importé, le système va le chercher, dans cet ordre :

- Parmi les modules du système Python (built-in)
- Dans le répertoire courant
- Dans les répertoires listés dans la variable d'environnement $PYTHONPATH
- Dans les répertoires par défaut de l'installation

Remarque : la liste des répertoires est donnée dans la variable `sys.path`, modifiable par le programme.

Variables globales

Définition

Une **variable globale** est une variable faisant partie d'un script ou d'un module et ayant portée dans tout ce script ou ce module. En particulier, elle peut être utilisée (en lecture ou en écriture) dans toutes les fonctions et procédures définies dans ce script ou ce module.

En Python, une variable globale doit :

- Être utilisée au moins une fois au plus haut niveau d'exécution dans le script
  - Pour être vue de toutes les fonctions appelées dans ce script
- Être déclarée comme globale, avec le mot-clé `global`, quand elle est utilisée dans une fonction

Pour déclarer une variable globale, on la met généralement

- Au plus haut niveau d'indentation
- En haut du script, avant les déclarations de fonctions

Intérêt : lisibilité, regroupement des variables globales, diminution du risque d'erreurs

Utilisation d'une variable globale

Pour utiliser une variable globale dans une fonction :

- **Toujours** la déclarer comme globale, avec le mot-clé **global**

```
1  def maFonction():
2      global varGlob
3      print varGlob
```

- Sinon elle sera considérée comme une variable locale

Exemple :

```
1  #!/usr/bin/python
2
3  globVar = 0
4
5  def fonction():
6      global globVar
7      globVar = 5
8
9  def main():
10     global globVar
11     print globVar
12     globVar = 1
13     print globVar
14     fonction()
15     print globVar
16
17 if __name__ == "__main__":
18     main()
```

On déclare une variable globale `globVar`, initialisée à 0

- La fonction `main()` déclare utiliser la variable globale `globVar` et la modifie
- Puis la fonction `fonction()` l'utilise et la modifie également

Affichage obtenu :

```
1  0
2  1
3  5
```

Paramètres optionnels de fonctions

Python permet d'utiliser des paramètres de fonctions **optionnels** et des paramètres **nommés**

Exemple :

```
1 def maFonction( a, b = 1, c = 0 ):
2     return a + b + c
```

Ici, les paramètres b et c sont optionnels : ils ont respectivement la valeur 1 et 0 par défaut. On peut appeler la fonction de plusieurs façons :

```
1 maFonction( 2, 4, 6 )
2 maFonction( 1, 2 )      # ici b vaut 2
3 maFonction( 1, c = 2 ) # ici c vaut 2
```

Remarque : vous avez vu quelques fonctions fonctionnant de cette façon. Par exemple la fonction `range()` utilise des arguments optionnels.

Attention : les arguments obligatoires doivent **toujours** être situés **avant** les arguments optionnels.

La remontée d'évènements dans un programme

But : signaler un évènement survenu dans une fonction

- Mécanisme d'**exception**
- Lorsqu'un évènement survient, on **soulève une exception**

Par exemple :

- Une erreur
- Une condition validée
- Un cas particulier rencontré

Une exception permet de **remonter** un évènement survenu.

Lever une exception

Utilisation du mot-clé **raise**

- On **soulève une exception donnée**
- D'un type donné

Par exemple, exceptions fournies par le système :

- IOError : erreur d'entrée-sortie
- OverflowError : débordement de tampon
- StopIteration : pas besoin d'aller plus loin
- TypeError : mauvais type utilisé

Exemple de levée d'exception

Petit exemple de levée d'exception

- On reprend l'exemple du calcul de quotient
- On ne peut pas diviser par zéro
  - On vérifie si le diviseur est nul
  - Si c'est le cas → levée de l'exception `ZeroDivisionError`

```
1  def quotient( dividende, diviseur ):
2      if diviseur == 0:
3          raise ZeroDivisionError
4      else:
5          return float( dividende ) / float( diviseur )
```

Utilisation du mot-clé **raise** suivi du **nom de l'exception soulevée**

- On ne cherche pas à aller plus loin
- Quand l'exception est soulevée, l'exécution de la séquence d'instructions est stoppée
- L'exception est **signalée à l'appelant**

Attraper une exception

Toute exception levée doit être attrapée (sinon cela ne servirait à rien)

La séquence d'instructions susceptible de lever une exception est exécutée dans un bloc particulier : le bloc **try**

- Mot-clé **try**
- Suivi des deux points (:)
- Le bloc est indenté d'un cran vers la droite

```
1 try:
2     c = quotient( a, b )
3     print c
```

Si une exception est soulevée, l'exécution de la séquence d'instructions située dans le bloc est **arrêtée**

- Dans l'exemple, si la fonction `quotient()` soulève une exception, on ne passe pas dans le `print`

Traitement des exceptions

Directement **à la suite du bloc `try`** on met les instructions de gestion des exceptions

- Définissent le comportement à suivre si une exception est soulevée

Les exceptions soulevées sont traitées dans des clauses **except**

```
1 try:
2     bloc d'instructions
3 except:
4     traitement des exceptions
```

On peut avoir plusieurs clauses except

- Différentes actions à effectuer en fonction de l'exception soulevée

On précise alors le type d'exception soulevée :

```
1 try:
2     bloc d'instructions
3 except Exception1:
4     traitement des exceptions de type Exception1
5 except Exception2, Exception3:
6     traitement des exceptions de type Exception2 ou Exception3
7 except:
8     traitement des exceptions des autres types
```

Mot-clé except tout seul : tous types d'exceptions

Traitement des exceptions

Si aucune exception n'a été soulevée : bloc **else**

```
try:
    bloc d'instructions
except Exception1:
    traitement des exceptions de type Exception1
except:
    traitement de tous les autres types d'exceptions
else:
    traitement si aucune exception n a ete soulevee
```

Bloc toujours exécuté : mot-clé **finally** :

```
try:
    bloc d'instructions
except:
    traitement des exceptions
finally:
    bloc toujours execute
```

Informations remontées par les exceptions

Intérêt des exceptions : savoir ce qui s'est mal passé

- Valeurs remontées dans l'exception
- On peut lui passer des paramètres en l'appelant (dépend de la définition de l'exception)

Exemple : l'exception `TypeError` peut prendre en paramètre la variable concernée

```
1     raise TypeError( var )
```

La valeur peut alors être récupérée par l'appelant qui attrape l'exception

- Informations fournies dans une fonction et des variables du module `sys`
- `exc_info()` contient un tuple donnant des informations : type, valeur et pile d'appels
- `exc_value` contient la valeur remontée par l'exception
- `exc_type` contient le type d'exception soulevée

Créer des nouveaux types d'exceptions

Le système fournit un certain nombre d'exceptions prédéfinies

- On peut vouloir écrire un type particulier d'exception
- On doit alors définir une nouvelle **classe d'exceptions**

NB : la terminologie de "classe" provient de la programmation orientée objet

Définition d'une nouvelle classe d'exception : mot-clé **class**

- Suivi du nom de la classe
- Et entre parenthèses : la catégorie d'exceptions dont elle fait partie (généralement `Exception`, sinon plus spécifique)

Exemple :

```
1 class ErreurMonException( Exception ):
2     definition de ma classe d'exception
```

Contenu d'un nouveau type d'exception

Une classe d'exception doit définir deux fonctions :

- `__init__()` : procédure d'initialisation de l'entité exception
- `__str__()` : fonction qui permet à `print` d'afficher l'état de l'exception sous forme d'une chaîne de caractères. Retourne une chaîne de caractères.

Au moins un paramètre obligatoire : **self**

On peut définir plusieurs fonctions `__init__()`, suivant les paramètres que l'exception peut prendre :

```
1  class ErreurMonException( Exception ):
2
3      def __init__( self ):
4          self.str = "Attention erreur"
5
6      def __init__( self, nb ):
7          self.str = "erreur sur le nombre: "
                  + str( nb )
8
9      def __str__( self ):
10         return self.str
```

Appeler notre nouveau type d'exception

Dans l'exemple, nous avons défini deux fonctions `__init__()` :

- Une fonction ne prenant pour argument que `self`
- Une fonction prenant pour argument `self` et un argument `nb` : c'est le **paramètre qui peut être passé lorsque l'exception est soulevée**

Deux possibilités pour soulever notre exception :

```
1 raise ErreurMonException( nb )
2 raise ErreurMonException
```

Manipulation de fichiers

La manipulation de fichiers se fait *toujours* en **trois étapes**:

- Ouverture du fichier
- Manipulation du fichier
- Fermeture du fichier

Attention à ne pas oublier d'ouvrir le fichier !

Attention à ne pas oublier de fermer le fichier !

Attention à bien gérer les erreurs :

- Fichier inexistant
- Disque plein donc écriture impossible
- Pas les bons droits sur le fichier
- ... (plein de possibilités d'erreurs)

Donc : gérer les exceptions !

Ouverture de fichier

Fonction **open()**

- Deux arguments :
  - **Chemin vers le fichier** (chaîne de caractères)
  - **Mode d'ouverture** (chaîne de caractères)
- Retourne un descripteur de fichier

Ouverture du fichier `/etc/hosts` en lecture :

```
1 fd = open( '/etc/hosts', 'r' )
```

On récupère le descripteur de fichiers `fd` :

```
1 >>> type( fd )
2 <type 'file'>
3 >>> print fd
4 <open file '/etc/hosts', mode 'r' at 0
      x7facf5f3f390>
```

C'est sur ce **descripteur de fichier** que l'on va effectuer les manipulations sur le fichier.

Mode d'ouverture de fichier

Le mode d'ouverture précise :

- Si on ouvre en lecture, en écriture, ou en lecture/écriture
- Où on se place initialement dans le fichier : début ou fin
- Si le fichier est tronqué à zéro ou non

Mode	Utilisation
r	Lecture
w	Écriture (créé ou tronqué à 0)
r+	Mise à jour (lecture et écriture)
w+	Tronque le fichier à zéro et l'ouvre en lecture-écriture
a	Ajoute à la fin du fichier
a+	Lit au début du fichier, ajoute à la fin du fichier

- Avec `r` ou `w` : on est positionné au **début** du fichier
- Avec `a` : on est positionné à la **fin** du fichier (*append*)
- Avec `w` : on tronque le fichier à 0 ou on le crée
- Pour écrire sans écraser : `a` ou `a+` à la fin, `r+` au début

Exceptions possibles à l'ouverture

Si on le peut pas ouvrir le fichier : exception **IOError** soulevée

- Plus de détails dans **sys.exc_value**

```
1  try:
2      fd = open( '/var/log/syslog', 'r' )
3  except:
4      import sys
5      print "Erreur ouverture", sys.exc_type, sys.exc_value
```

On n'a pas les droits en lecture sur ce fichier. Résultat :

```
1  Erreur ouverture <type 'exceptions.IOError'> [Errno 13] Permission denied: '/var/
       log/syslog'
```

Fermeture d'un fichier

Fonction **close()** sur le descripteur de fichier :

```
1 fd.close()
```

- On peut fermer une fichier plus d'une fois
- On ne peut pas fermer un fichier qui n'a pas été ouvert
- On doit fermer un fichier quand on en a fini avec lui

Des tentatives d'opérations sur un fichier fermé soulèveront l'exception **ValueError**

Lecture dans un fichier

Plusieurs possibilités de lecture dans un fichier :
Si c'est un fichier texte : lecture **ligne par ligne**

- Fonction **readline()** sans argument
- Retourne une chaîne de caractères

```
1 ligne = fd.readline( )
2 print "ligne lue : ", ligne
```

Si c'est un **fichier binaire** : lecture **octet par octet**

- Fonction **read()** avec comme argument le nombre d'octets à lire ou rien pour tout lire d'un coup
- Retourne une chaîne de caractères
- Si on en lit moins (fichier plus court) : `len()` de ce qui est retourné pour avoir la taille réelle

```
1 ret = fd.read( 512 )
2 print "lus : ", len( ret )
```

Lecture de tout un fichier entier

Si c'est un fichier texte : lecture de **toutes les lignes**

- Fonction **readlines()** sans argument
- Retourne une liste de chaînes de caractères
- On parcourt ensuite cette liste

```
1 lignes = fd.readlines( )
2 for l in lignes:
3     print l
```

Si c'est un **fichier binaire** : lecture **de tous les octets**

- Fonction **read()** sans **aucun argument**
- Retourne une chaîne de caractères qui contient tout le contenu du fichier

```
1 ret = fd.read( )
2 print "lus : ", len( ret )
```

Fin de fichier

Quand on arrive à la **fin du fichier**, les fonctions `read()` et `readline()` retournent une chaîne vide :

```
1 line = fd.readline()
2 while line:
3     print line
4     line = fd.readline()
```

Avec `read()` en passant la longueur à lire en argument, si on atteint la fin du fichier on s'arrête là et on obtient la longueur réellement lue avec `len()` :

```
1 buffsize = 512
2 ret = fd.read( buffsize )
3 lu = len( ret )
4 while len( ret ) == buffsize:
5    ret = fd.read( buffsize )
6    lu += len( ret )
7 print lu, "octets lus"
```

Écriture dans un fichier

On écrit de la même façon du texte ou des données binaires

- Écrire une **chaîne de caractères ou d'octets** : procédure **write()**

```
1 fd.write( "texte" )
```

- Écrire une **liste d'éléments** : procédure **writelines()**

Ce sont des **procédures** : elles ne retournent rien.

Exceptions soulevées lors des manipulations de fichiers

Attention aux exceptions soulevées

- Très utiles ! Et très courantes
- Système de fichier plein, droits inadaptés, tentative d'écriture sur un descripteur de fichier fermé...

Exemple : on essaye d'écrire sur un fichier qui a été fermé

```
1  fd.close()
2  try:
3      r = fd.read()
4  except:
5      import sys
6      print "Erreur de lecture", sys.exc_type, sys.exc_value
```

Donnera :

```
1  Erreur de lecture <type 'exceptions.ValueError'> I/O operation on
       closed file
```

Autres fonctions utiles : `seek()`

On peut se **déplacer dans un fichier** : procédure **seek()**

- On spécifie le nombre d'octets dont on se déplace
- Optionnel : point de référence du déplacement
  - 0 : par rapport au début du fichier (valeur par défaut)
  - 1 : par rapport à la position actuelle
  - 2 : par rapport à la fin du fichier

```
1 fd.seek( 0 )       # on se place au debut du fichier
2 fd.seek( 0, 2 )    # on se place a la fin du fichier
3 fd.seek( 512, 1 ) # on avance de 512 octets
4 fd.seek( -64, 1 ) # on recule de 64 octets
```

Autres fonctions utiles : `flush()`

Optimisation de l'OS : les écritures sur disque sont mises en cache dans des tampons internes Python et de l'OS

- Conséquence : ce qui est écrit dans les fichiers n'est pas forcément mis sur le disque immédiatement
- Peut poser des problèmes (accès concurrents, programme qui plante...)

On peut forcer le vidage du tampon interne avec **flush()**

- ... mais `flush()` ne force pas l'écriture sur disque :-(
- Procédure du module `os` : **os.fsync()**
- On lui passe le descripteur de fichier

```
1 fd.flush()
2 os.fsync(fd_out)
```