# Algorithmique et Informatique

**M 1204**

1re année, IUT du Limousin

Département Mesures Physiques


Ludovic GROSSARD

ludovic.grossard@unilim.fr

M1204 – Algorithmique et informatique

# Chapitre 1 : présentation

Ludovic Grossard

Département Mesures Physiques, IUT du Limousin
Université de Limoges





## I. Définitions

### Algorithme

ensemble des règles opératoires qui permettent la résolution d'un problème par l'application d'un nombre fini d'opérations de calcul à exécuter en séquence.

### Pseudo-code

façon de décrire un algorithme sans référence à un langage de programmation en particulier. Il n'existe pas de réelle convention pour le pseudo-code.

# II. La recette de cuisine est un algorithme

## Vous avez des algorithmes à la maison

**Ingrédients pour 60 petits biscuits :**

- 50 g d'huile de palme en pain
- 50 g de vergeoise belge Graeffe (ou sucre)
- 200 g de farine de riz
- 1/2 cuillère à thé de bicarbonate de sodium
- 1 c. à soupe de citron pressé
- 5 a 7cl de lait de riz ou jus de fruit
- chocolat noir « sans » pour allergique

**Sablé nappé de chocolat**

Mélangez tous ces ingrédients dans un robot ou à la main, en ajoutant peu à peu le liquide jusqu'à former une boule. Laissez reposer cette pâte une heure dans un endroit frais (mais on peut se passer de cette étape, puisqu'il n'y a pas de levure dans la pâte).


Tout comme une recette de cuisine, un algorithme est caractérisé par :

- 
- 
- 





# III. La recette n'est pas un algorithme

**Ingrédients pour 60 petits biscuits :**

- 50 g d'huile de palme en pain
- 50 g de vergeoise belge Graeffe (ou sucre)
- 200 g de farine de riz
- 1/2 cuillère à thé de bicarbonate de sodium
- 1 c. à soupe de citron pressé
- 5 a 7cl de lait de riz ou jus de fruit
- chocolat noir « sans » pour allergique

**Sablé nappé de chocolat**

Mélangez tous ces ingrédients dans un robot ou à la main, en ajoutant peu à peu le liquide jusqu'à former une boule. Laissez reposer cette pâte une heure dans un endroit frais (mais on peut se passer de cette étape, puisqu'il n'y a pas de levure dans la pâte).

- Une recette est ambigüe (termes vagues, plusieurs options)
- nécessite également le « coup de main » du chef
- ne mène pas toujours au résultat attendu

# IV. Pourquoi apprendre l'algorithmique ?

## et pas apprendre directement un langage de programmation

- L'algorithme exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage ;
  Il est écrit en
- permet d'apprendre à manier la structure logique d'un programme informatique quel que soit le langage de programmation ;
- vous pourrez ensuite passer facilement d'un langage à un autre.



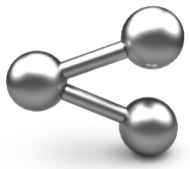


# V. Les briques de base

Un algorithme, aussi complexe soit-il, est composé de quatre familles d'instructions :


l'affectation de variables


la lecture / écriture



les tests

les boucles

# VI. De l'algorithme au programme

- L'algorithme ne représente que la suite des opérations à effectuer,
- sa mise en œuvre se fait via un ,
- le résultat de cette transcription s'appelle le

Il existe un grand nombre de langages de programmation (plus de 700 selon wikipedia), chacun ayant ses spécificités.

Nous utiliserons le langage

M1204 – Algorithmique et informatique

# Chapitre 2 : les variables

Ludovic Grossard

Département Mesures Physiques, IUT du Limousin
Université de Limoges





## I. Définitions

### Variable

- permet au programme de travailler sur des données
- et de stocker le résultat
- les variables sont indispensables en programmation
- leur emplacement dans la mémoire s'appelle

  mais on les désigne généralement par un

- attention, la mémoire est volatile (son contenu disparaît à la fin du programme)

support physique :

# II. Types de données

Les variables peuvent accueillir des données simples :

- 
- 

et des structures de données plus complexes :

- 
- 
- 





# III. Noms des variables

- Chaque variable possède un nom, choisi par le programmeur

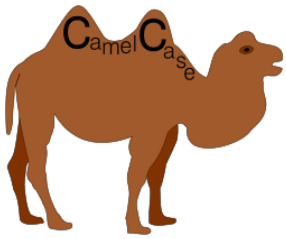
Quelques règles :

- le nom de la variable doit indiquer de manière claire son rôle (prixBoisson, monnaieIntroduite. . .)
- les compteurs sont généralement nommés avec une seule lettre : i, j, k. . .
- pas d'espaces, pas d'accents ou caractères spéciaux ($#!@«». . .) sauf le caractère de soulignement (underscore _)
- on évite les noms compliqués ou sans signification : toto, zz42, nbPtsdsplan (difficiles à retenir, risque d'erreur de saisie)
- ne pas hésiter à prendre un nom long si besoin : vitesseMoyenneEnkmh

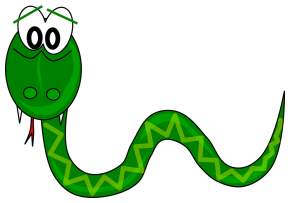
# III. Noms des variables

Conventions généralement utilisées pour nommer les variables

`vitesseMoyenneEnkmh`




`vitesse_moyenne_en_kmh`



Une fois la convention fixée, il faut s'y tenir tout au long du projet.





# IV. Déclaration des variables

Au début de chaque algorithme, on déclare les variables utilisées et le type de données qu'elles accueilleront

```
VARIABLES
  monnaieIntroduite DE_TYPE nombre
  numeroBoisson DE_TYPE nombre

  ...
```

- Certains langages nécessitent que les variables soient déclarées
- ce n'est pas le cas du langage Python
- mais nous les déclarerons en commentaires

```
1 # variables
2 # monnaieIntroduite DE_TYPE nombre
3 # numeroBoisson DE_TYPE nombre
4 ...
```

# V. Affectation à une variable

Pour donner une valeur à une variable, on utilise l'

- L'expression à droite du symbole ← est évaluée, puis placée dans la variable située à gauche.
- on peut (ré)affecter une variable à n'importe quel moment
- Cette expression peut être :
  - une constante
  - une autre variable
  - une expression

Exemple : l'incrémentation :





# V. Affectation à une variable

**Attention**

Avant de pouvoir utiliser une variable dans un algorithme, elle doit être initialisée

```
VARIABLES
  a DE_TYPE nombre
  b DE_TYPE nombre
  c DE_TYPE nombre

DEBUT_ALGORITHME
  a ← 1
  c ← a + b
  # Problème ! Que vaut b ?

FIN_ALGORITHME
```

# VI. Opérateurs

## 1) Opérateurs et opérandes

**opérateur**

**opérande**

**expression**

Par exemple, dans l'expression :
x ← y + 1 Il y a opérateurs et opérandes



Définitions Types Noms Déclaration Affectation Les opérateurs Lire et afficher Exemple

VI. Opérateurs

1) Opérateurs et opérandes

Il existe différents types d'opérateur :

- d'affectation
- mathématiques
- de comparaison
- logiques

# VI. Opérateurs

## 2) Opérateurs mathématiques

Les opérateurs suivants sont disponibles :

| | | | |
|---|---|---|---|
| + | addition | x ← y + z | |
| - | soustraction | x ← y - z | |
| * | multiplication | x ← y * z | |
| ** | élévation à la puissance | x ← y ** 3 | |
| / | division | x ← y / z | attention à la division par 0 ! |
| % | modulo | x ← y % z | y et z entiers |

exemple :





# VI. Opérateurs

## 3) Priorité des opérateurs

de la priorité la plus forte à la plus faible :

| | |
|---|---|
| ** | exponentiation |
| *, /, % | multiplication, division et reste |
| +, - | addition et soustraction |

Les opérateurs avec la même priorité sont listés dans la même ligne.
Par exemple, + et − ont la même priorité.

**Exemple**

x ← 5 + 3 * 2**2

Pour modifier l'ordre d'évaluation, on utilise des parenthèses :
x ← (5 + 3) * 2**2

# VII. Lire et afficher des variables

Un programme doit pouvoir dialoguer avec l'utilisateur

lire les données qu'il saisit au clavier

afficher des informations à l'écran

Remarque :

- lorsque le programme lit des données, l'utilisateur écrit.
- Lorsque le programme affiche des données, l'utilisateur les lit.





# VII. Lire et afficher des variables

- LIRE demande à l'utilisateur de saisir au clavier un nombre ou du texte et le place dans une variable

```
LIRE  x
```

- AFFICHER permet d'afficher à l'écran :
  - soit du texte (entre guillemet)
  - soit le contenu d'une variable

```
AFFICHER  "la valeur de x est : "
AFFICHER  x
```

# VIII. Exemple

## Calcul d'un bénéfice

Écrire un algorithme qui calcule le bénéfice mensuel réalisé par un transporteur

**Recettes**

- 200 voyageurs par jour
- à 2€ TTC le ticket
- et une TVA à 7%
- jours ouvrés dans le mois : 20

**Dépenses**

- 600 km par jour
- consommation : 10 l/100 km
- prix du litre : 1.5 €
- coût d'entretien journalier : 50€





# VIII. Exemple

## Calcul du jour de la semaine

Cet algorithme permet de calculer le jour de la semaine pour une date donnée dans le calendrier Grégorien. Cette méthode de calcul est valable pour les dates à partir du 1er novembre 1582.

Pour une date de la forme `jour / mois / annee` où `jour` prend une valeur de 01 à 31, `mois` de 01 à 12 et `annee` de 1583 à 9999, on utilise la formule suivante :

```
j = ( jour + a + a/4 - a/100 + a/400 + 31*m/12 ) % 7
```

avec

- `c = (14 - mois)/12` (c = 1 pour janvier et février, c = 0 pour les autres mois)
- `a = annee - c`
- `m = mois + 12*c - 2`
- % signifie modulo (reste de la division entière)

La réponse obtenue pour `j` correspond alors à un jour de la semaine suivant :
0 = dimanche, 1 = lundi, 2 = mardi, etc.

M1204 – Algorithmique et informatique

# Chapitre 3 : les structures de contrôle

Ludovic Grossard

Département Mesures Physiques, IUT du Limousin
Université de Limoges





## I. Définition

- Les instructions d'un algorithme sont exécutées dans leur ordre d'apparition
- cependant, un algorithme doit souvent évaluer des conditions, et prendre des décisions
- il existe pour cela les structures de contrôles

### Structure de contrôle

Deux types de structures de contrôles :

-
-

# II. Opérateurs de comparaison

Ils permettent d'effectuer des tests.

| | |
|---|---|
| == | égal à |
| != | différent de |
| < | inférieur à |
| <= | inférieur ou égal à |
| > | supérieur à |
| >= | supérieur ou égal à |

Le résultat d'une comparaison est de type booléen et vaut :

- si la condition est vérifiée
- si elle ne l'est pas

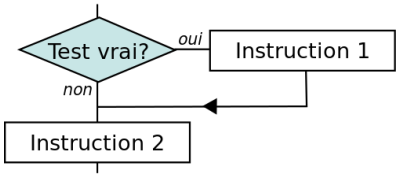




# III. Les alternatives

## 1) Test si (if)

```
SI (Test) ALORS
    DEBUT_SI
    # instruction 1
    FIN_SI
# instruction 2
# ...
```




- si `Test` est vérifié on exécute Instruction 1 puis Instruction 2
- si `Test` n'est pas vérifié on exécute directement Instruction 2
- `Test` est un . Il vaut `vrai` ou `faux`.

# III. Les alternatives

## Test si : exemple

Le pseudo-code suivant demande deux nombres entiers x et y à l'utilisateur, et affiche si x est multiple de y ou non.

```
VARIABLES
    x DE_TYPE NOMBRE
    y DE_TYPE NOMBRE
DEBUT_ALGORITHME
    LIRE x
    LIRE y
    SI (x % y == 0) ALORS
        DEBUT_SI
        AFFICHER x
        AFFICHER " est multiple de "
        AFFICHER y
        FIN_SI
FIN_ALGORITHME
```

Problème : rien n'est affiché si x n'est pas multiple de y !



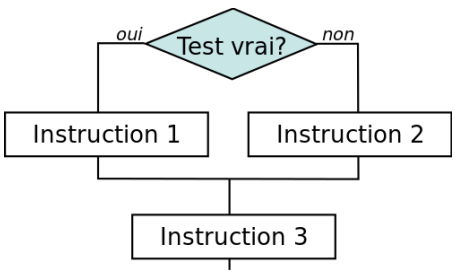


# III. Les alternatives

## 2) Test si sinon (`if else`)

Le test **si sinon** permet d'exécuter également des instructions si le test n'est pas vérifié

```
SI (Test) ALORS
    DEBUT_SI
    # instruction 1
    FIN_SI
    SINON
        DEBUT_SINON
        # instruction 2
        FIN_SINON
# instruction 3
# ...
```




- si `Test` est vérifié, on exécute instruction1 (puis instruction3)
- sinon, on exécute Instruction 2 (puis Instruction 3)

# III. Les alternatives

## Test si sinon : exemple

Version modifiée avec un test **si** puis **sinon** :

```
VARIABLES
    x DE_TYPE NOMBRE
    y DE_TYPE NOMBRE
DEBUT_ALGORITHME
    LIRE x
    LIRE y
    SI (x % y == 0) ALORS
        DEBUT_SI
        AFFICHER x
        AFFICHER " est multiple de "
        AFFICHER y
        FIN_SI
        SINON
            DEBUT_SINON
            AFFICHER x
            AFFICHER " n'est pas multiple de "
            AFFICHER y
            FIN_SINON
FIN_ALGORITHME
```



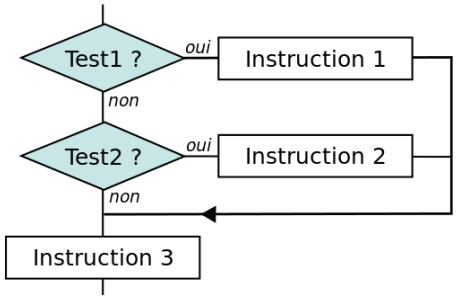


# III. Les alternatives

## 3) Test sinon si (`if elif`)

Le test **si sinon si** permet d'enchaîner des tests si

```
SI (Test) ALORS
    DEBUT_SI
    # instruction 1
    FIN_SI
    SINON
        DEBUT_SINON
        SI (Test 2) ALORS
            DEBUT_SI
            # instruction 2
            FIN_SI
        FIN_SINON
# instruction 3
# ...
```




- On peut enchaîner autant d'instructions `sinon si` que désiré : seule la première dont la condition sera vérifiée sera exécutée.
- On peut généralement associer une clause `sinon` qui sera exécutée uniquement si aucune clause `sinon si` n'a été vérifiée (traitement par défaut).

# IV. Les opérateurs logiques

## 1) L'opérateur logique ET (`and`)

Permet de tester si deux conditions sont vraies

```
SI (condition 1 ET condition 2) ALORS
    DEBUT_SI
    # ...
    FIN_SI
```

Le test vaut vrai si
sinon le test vaut faux.





# IV. Les opérateurs logiques

## 2) L'opérateur logique OU (`or`)

Permet de tester si au moins une de deux conditions est vraie.

```
SI (condition 1 OU condition 2) ALORS
    DEBUT_SI
    # ...
    FIN_SI
```

Le test vaut vrai
sinon le test vaut faux.

# IV. Les opérateurs logiques

## 3) L'opérateur logique NON (`not`)

Permet de tester si une condition est fausse

```
SI (NON condition) ALORS
    DEBUT_SI
    # ...
    FIN_SI
```

L'opérateur `NON` renvoie `vrai` si son opérande est `faux`, et inversement

Permet d'exécuter les instructions





# IV. Les opérateurs logiques

## 4) Tables de vérité

| C1 ET C2 | C2 vrai | C2 faux |
|---|---|---|
| C1 vrai | vrai | faux |
| C1 faux | faux | faux |

| C1 OU C2 | C2 vrai | C2 faux |
|---|---|---|
| C1 vrai | vrai | vrai |
| C1 faux | vrai | faux |

| NON C1 | |
|---|---|
| C1 vrai | faux |
| C1 faux | vrai |

# V. Les boucles

## 1) Définition

### Boucle





# V. Les boucles

## 2) La boucle Tant que ( `while`)

```
TANT_QUE (Test) FAIRE
    DEBUT_TANT_QUE
    # instructions
    FIN_TANT_QUE
```

Principe :

- Le programme examine la valeur du `Test`
- s'il vaut `vrai`, les instructions qui suivent sont exécutées
- jusqu'à ce qu'il rencontre la ligne **`FIN_TANT_QUE`**
- il retourne à la ligne du **`TANT_QUE`**, et ainsi de suite
- L'exécution des intructions ne s'arrête que lorsque le `test` vaut `faux`

# V. Les boucles

## 3) Boucle `Tant que` contrôlée par un compteur

Permet d'effectuer des boucles dont on connaît à l'avance le nombre d'itérations. Elle exige :

- 
- 
- 
- 

Modification courante du compteur :

$$i \leftarrow i+1$$





# V. Les boucles

## 3) Boucle `Tant que` contrôlée par un compteur

pseudo-code :

```
VARIABLES
    i DE_TYPE NOMBRE # variable de contrôle
DEBUT_ALGORITHME
    # initialisation du compteur
    TANT_QUE (condition) FAIRE
        DEBUT_TANT_QUE
        # instructions
        # ...
        # incrémentation du compteur
        FIN_TANT_QUE
FIN_ALGORITHME
```

1. L'initialisation doit avoir lieu
2. la condition doit porter
3. l'incrémentation se fait généralement

# V. Les boucles

## 3) Boucle `Tant que` contrôlée par un compteur

Exemple : affichage des nombres de 1 à 10 :

```
VARIABLES
    i DE_TYPE NOMBRE # compteur
DEBUT_ALGORITHME
    i ← 1
    TANT_QUE (i<=10) FAIRE
        DEBUT_TANT_QUE
        AFFICHER i
        i ← i + 1
        FIN_TANT_QUE
FIN_ALGORITHME
```

- On peut également utiliser la condition
- que se passe-t-il si on place l'incrémentation avant les instructions ?
- que se passe-t-il si on oublie l'incrémentation ?





# V. Les boucles

## 4) Boucle Pour (`For`)

Affichage des nombres de 1 à 10 :

```
VARIABLES
    i DE_TYPE NOMBRE # compteur
DEBUT_ALGORITHME
    POUR i ALLANT_DE 1 A 10
        DEBUT_POUR
        AFFICHER i
        FIN_POUR
FIN_ALGORITHME
```

Il est inutile :

- d'initialiser le compteur
- de l'incrémenter

# V. Les boucles

## 5) Boucles imbriquées

### Définition

Pour chaque itération de la première boucle, la deuxième sera exécutée en entier.

```
TANT_QUE (condition 1) FAIRE
    DEBUT_TANT_QUE
    TANT_QUE (Condition 2) FAIRE
        DEBUT_TANT_QUE
        # instructions
        # ...
        FIN_TANT_QUE
    FIN_TANT_QUE
```

M1204 – Algorithmique et informatique

# Chapitre 4 : les listes

Ludovic Grossard

Département Mesures Physiques, IUT du Limousin
Université de Limoges





## I. Définition

### liste (ou tableau)

liste

| elt0 | elt1 | elt2 | elt3 | ... |
|---|---|---|---|---|

Chaque élément dans la liste est accessible grâce à son

- le premier élément de la liste correspond à l'indice `0`
- Pour une liste de `N` éléments, les indices vont de `0` à `N-1`

# II. Création d'une liste

## 1) Initialisation directe

```
VARIABLES
  notes DE_TYPE LISTE # inutile d'indiquer le nombre d'éléments

DEBUT_ALGORITHME
  notes[0] ← 12.5 : 8.5 : 16 : ...

FIN_ALGORITHME
```

- On sépare les éléments par des caractères « deux points »
- et on indique que les données sont placées à la position 0 dans la liste

Modification d'un élément :

```
notes[1] ← 10.0
```

La liste contient alors les valeurs `12.5:10.0:16:...`





# II. Création d'une liste

## 1) Exemple

On demande à l'utilisateur de saisir 5 nombres, qui sont placés dans une liste.

```
VARIABLES
    nombres DE_TYPE LISTE
    i DE_TYPE NOMBRE # compteur
    x DE_TYPE NOMBRE # chaque nombre saisi
DEBUT_ALGORITHME
    # boucle pour saisir les 5 nombres
    POUR i ALLANT_DE 0 A 4
        DEBUT_POUR
        LIRE x
        nombres[i] ← x
        FIN_POUR
FIN_ALGORITHME
```

# III. Accès aux donnés d'une liste

## 1) Accès à un élément

on écrit le nom de la liste, suivi de l'indice de l'élément entre crochets :

```
notes ← 12.5 : 8.5 : 16


AFFICHER  notes[1]
# affiche la valeur : 8.5
```

Remarques :

- l'indice est obligatoirement un nombre entier
- ne confondez pas la valeur de l'indice (entier i), et le contenu de la liste pour cet indice.
- `notes[3]` provoquera une erreur

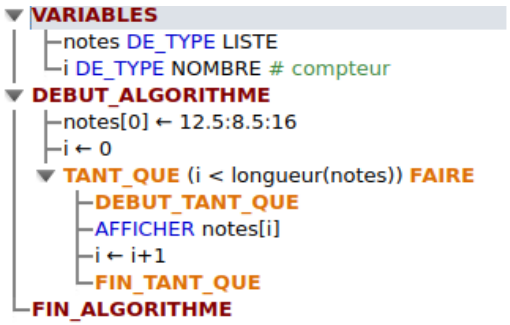




# III. Accès aux donnés d'une liste

## 2) Balayer tous les éléments

- on utilise une boucle
- on utilise la pour connaître le nombre d'éléments dans la liste



M1204 - Algorithmique et informatique

# Chapitre 5 : les fonctions

Ludovic Grossard

Département Mesures Physiques, IUT du Limousin
Université de Limoges

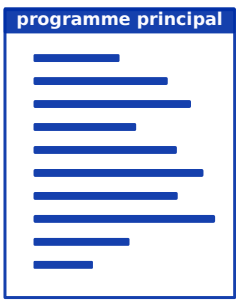


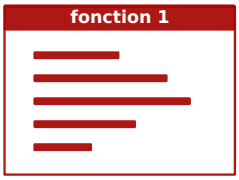


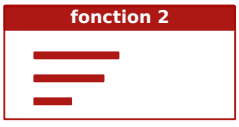
## I. Définition

### fonction

- les fonctions permettent de décomposer un problème compliqué en plusieurs problèmes plus simples
- le programme principal fait appel à des fonctions
- qui peuvent elles même faire appel à d'autres fonctions



# II. Fonctions prédéfinies

Les différents langages de programmation disposent d'un grand nombre de fonctions prédéfinies.

Ces fonctions sont caractérisées par :

- 
- 
  Ces parenthèses sont obligatoires, même si on ne met rien à l'intérieur
- une liste de
- Une fonction peut avoir aucun, un ou plusieurs paramètres.
  Exemple : la fonction `sin` nécessite un paramètre, l'angle en radians : `sin(1.234)`





# II. Fonctions prédéfinies

Les paramètres des fonctions peuvent être des variables :

```
a ← 1.234
AFFICHER  sin(a)
```

Les fonctions peuvent renvoyer un

Ce peut être :

- un nombre
- du texte
- une liste
- ou tout autre élément géré par le langage de programmation.

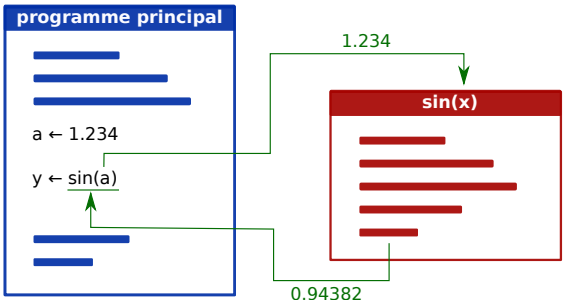
# II. Fonctions prédéfinies

le résultat peut être utilisé directement, soit placé dans une variable :

```
a ← 1.234
y ← sin(a)
```

Dans ce cas, pour affecter la valeur à y, le programme doit d'abord évaluer la fonction, c'est-à-dire :

1. 
2. 
3. 








# II. Fonctions prédéfinies

**Attention**

Ne jamais utiliser de variables portant le même nom que des fonctions.

Exemple

```
AFFICHER  sin(1.234)
# fonctionne
sin ← 2
# la fonction sin n'existe plus !
AFFICHER  sin
# affiche le nombre 2
AFFICHER  sin(1.234)
# provoque une erreur...
```

# III. Fonctions personnalisées

## 1) Écriture en pseudo-code

Elles permettent d'étendre un langage de programmation par ajout de nouvelles fonctions.

En pseudo-code :

```
FONCTION  nom_de_la_fonction ( paramètres )
    |
    |   Corps de la fonction
    |
FIN_FONCTION
```





# III. Fonctions personnalisées

## 2) Choix du nom de la fonction

le choix du nom est libre, sauf :

- mots réservés du langage (`SI`, `TANT QUE`, `POUR`...)
- caractères spéciaux ou accentués (caractère de soulignement _ autorisé)
- nom de fonction prédéfinie, à moins de vouloir les redéfinir
- écriture en snake_case (`nom_de_la_fonction`)

# III. Fonctions personnalisées

## 3) Renvoi du résultat

Lors de l'éxécution du programme, la fonction se termine :

- dès qu'elle rencontre une instruction
- ou qu'elle arrive à la fin de la fonction (dans ce cas, elle ne retourne rien)

Il peut y avoir plusieurs instructions `retourner` dans le corps de la fonction.

Une fonction ne peut renvoyer au plus qu'un résultat.

Si elle ne renvoie rien, on ne parle pas de fonction, mais de





# III. Fonctions personnalisées

## 4) Exemple de procédure

Pseudo-code d'une procédure qui affiche une table de multiplication :

```
PROCÉDURE affiche_table ( n )
    VARIABLES
      i DE_TYPE NOMBRE # compteur

    i ← 0
    TANT_QUE (i <= 10) FAIRE
        AFFICHER i , " * " , n , " = " , i*n
        i ← i + 1
    FIN_TANT_QUE
FIN_PROCÉDURE
```

`affiche_table(3)` afficherait :

```
0 * 3 = 0
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
...
10 * 3 = 30
```

# III. Fonctions personnalisées

## 5) Exemple de fonction

Fonction qui renvoie le plus grand des deux nombres passés en paramètre

```
FONCTION  maxi ( a , b )
    SI  (a > b ) ALORS
        retourner a
    SINON
        retourner b
    FIN_SI
FIN_FONCTION
```

```
AFFICHER maxi( 1.2 , 3.4 )
3.4
```

Remarques :

- une fonction peut contenir plusieurs instructions `retourner`
- ici, l'ordre des paramètres n'a pas d'importance





III. Fonctions personnalisées

5) Exemple de fonction

```
FONCTIONS
  FONCTION  maxi ( a , b )
      SI  (a > b ) ALORS
          retourner a
      SINON
          retourner b
      FIN_SI
  FIN_FONCTION

VARIABLES
  n1,n2 DE_TYPE NOMBRE
  maximum DE_TYPE NOMBRE

DEBUT_ALGORITHME

  AFFICHER  "n1 = "
  LIRE  n1
  AFFICHER  "n2 = "
  LIRE  n2

  maximum ← maxi(n1,n2)
  AFFICHER  "le maximum est : ",maximum

FIN_ALGORITHME
```

- `a` et `b` sont des paramètres
- se sont des variables à la fonctions, elles n'existent pas en dehors.
- `n1` et `n2` sont les paramètres
- `n1` est dans `a`, et `n2` dans `b`
- on pourrait écrire `AFFICHER maxi(n1,n2)` sans passer par une variable `maximum`

# III. Fonctions personnalisées

## 6) Remarques

- Le programme principal n'a pas besoin de connaître le fonctionnement interne de chaque fonction,
- Les seuls éléments à connaître sont :
  - les paramètres nécessaires (type, ordre),
  - le type de résultat renvoyé,
- ces informations doivent se trouver dans la documentation de la fonction,
- il est indispensable de documenter les fonctions que vous écrivez.

# Algorithmique et Informatique

**M 1204**

1re année, IUT du Limousin

Département Mesures Physiques

— TRAVAUX DIRIGÉS —

# Sommaire

Fiche de TD n° 1

# Variables, affectations

**Recommandations :** *Soyez rigoureux dans l'écriture des algorithmes en pseudo-code. Commencez par lister les variables utilisées en indiquant leur type et leur rôle si besoin. Définissez ensuite les fonctions s'il y en a, et enfin écrivez le contenu de l'algorithme. Indentez correctement votre pseudo-code pour en faciliter la lecture. Exécutez mentalement votre pseudo-code pour en vérifier le fonctionnement.*

## 1.1 Noms de variables

Pour chaque variable dont on vous donne la description, choisissez le nom qui vous semble le mieux adapté, en notation chamelCase puis en snake_case.

| Description de la variable | chamelCase | snake_case |
|---|---|---|
| une somme de tensions | | |
| la période d'une fonction | | |
| la célérité de la lumière | | |
| un compteur | | |
| la valeur moyenne d'une série de données | | |

## 1.2 Affectation de variables

Quelles sont les valeurs des différentes variables à la fin des algorithmes suivants :

**Algorithme 1**

```
VARIABLES
  a, b, c, d : entiers

DEBUT_ALGORITHME
  a ← 1
  b ← 3
  c ← b - 2a
  d ← c + a
  a ← d - b
  c ← c + 2
  b ← b - a
  a ← b

FIN_ALGORITHME
```

**Algorithme 2**

```
VARIABLES
  a, b, c, d : entiers

DEBUT_ALGORITHME
  a ← 1
  b ← 3
  c ← a + b
  a ← c + b
  b ← d - c
  b ← b - a

FIN_ALGORITHME
```

## 1.3 Une partie de golf

L'objectif est ici d'écrire un petit jeu qui simule une partie de golf. L'ordinateur choisit la distance séparant le joueur du trou, et le but du jeu est d'envoyer la balle le plus près du trou. Pour cela, le joueur choisit l'angle et la vitesse initiaux de la balle. Le programme calcule alors la distance à laquelle la balle touche le sol à partir des équations régissant sa trajectoire. Afin de simplifier le problème, vous ferez quelques approximations :

— les effets de la présence d'une atmosphère (frottements, vents...) ne sont pas pris en compte,
— la balle ne rebondit pas et ne roule pas lorsqu'elle touche le sol.

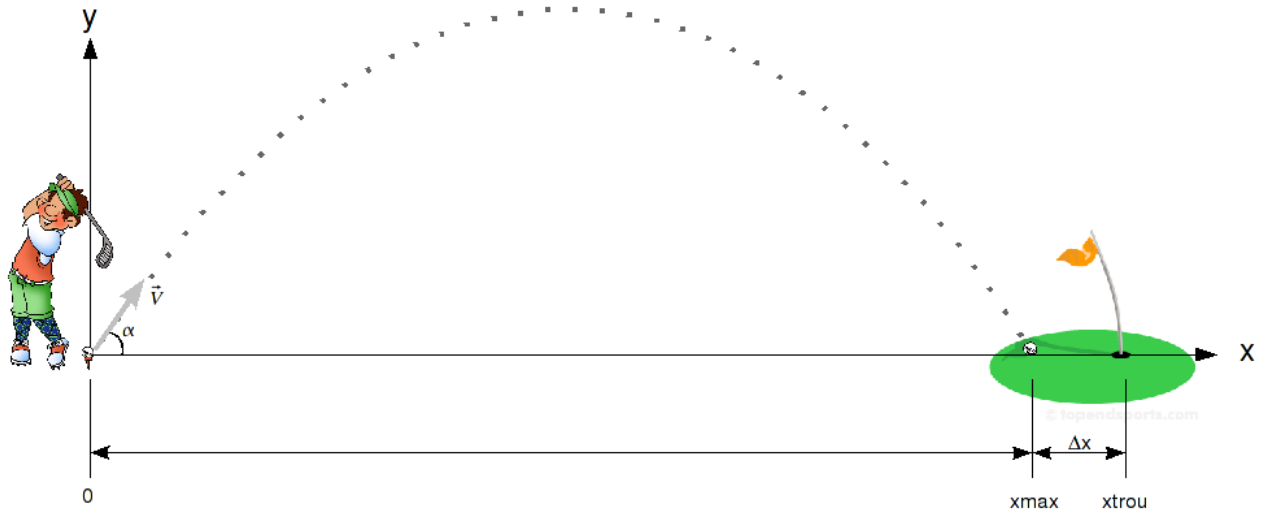



FIGURE 1.1 – Une petite partie de golf.

### 1.3.1 Étude préliminaire

Le système d'équations paramétriques décrivant la trajectoire de la balle s'écrit :

$$\begin{cases} x(t) = \|\vec{V}\| \cdot \cos(\alpha) \cdot t & (1.1) \\ y(t) = -\frac{1}{2} g t^2 + \|\vec{V}\| \cdot \sin(\alpha) \cdot t + y_0 & (1.2) \end{cases}$$

avec $g = 9,8\ \mathrm{m/s^2}$.

**1.1** ▷ En supposant que $y_0 = 0$, vérifiez qu'à l'instant initial $t = 0$ s, la balle se trouve bien à l'origine du repère.

**1.2** ▷ En utilisant l'équation (1.2), donnez l'expression du temps $t_{\text{vol}}$ au bout duquel la balle touche le sol. Vous rechercherez pour cela les racines du polynôme, et exprimerez $t_{\text{vol}}$ en fonction des coefficients $a$, $b$ et $c$ du polynôme, et du discriminant $\Delta$.

**1.3** ▷ En utilisant l'équation (1.1) et le résultat de la question précédente, déduisez-en la distance $x_{\text{max}}$ à laquelle se trouve alors la balle en fonction de $\|\vec{V}\|$, $\alpha$ et $t_{\text{vol}}$.

### 1.3.2 Programme de base

Dans un premier temps, on pose $y_0 = 0$ m

**1.4** ▷ Écrivez en pseudo-code l'algorithme du programme de base dont voici les différentes étapes :
— définissez les différentes variables utilisées dans le programme

— initialisez la variable `g`;
— demandez à l'utilisateur de saisir l'angle initial en degrés. Le programme placera la valeur saisie dans la variable `alpha_degres`;
— demandez ensuite la vitesse initiale de la balle en km/h. Le programme placera la valeur saisie dans la variable `v_kmh`;
— placez dans la variable `alpha_radians` l'angle initial en radians, calculé à partir de `alpha_degres`;
— de la même manière, placez dans la variable `v_ms` la vitesse initiale en m/s calculée à partir de `v_kmh`;
— initialisez alors les coefficients `a`, `b` et `c` du polynôme qui décrit la trajectoire de la balle suivant $y$;
— calculez le discriminant de ce polynôme et placez-le dans la variable `delta`
— calculez `tvol` et `xmax`;
— affichez enfin `xmax` à l'écran, sans oublier son unité.

Fiche de TD n° 2

# Tests et boucles

**Recommandations :** *Soyez rigoureux dans l'écriture des algorithmes en pseudo-code. Commencez par lister les variables utilisées en indiquant leur type et leur rôle si besoin. Définissez ensuite les fonctions s'il y en a, et enfin écrivez le contenu de l'algorithme. Indentez correctement votre pseudo-code pour en faciliter la lecture. Exécutez mentalement votre pseudo-code pour en vérifier le fonctionnement.*

## 2.1 Boucles simples

Écrivez les algorithmes en pseudo-code permettant d'afficher à l'écran les suites numériques suivantes :

**2.1** ▷ `1 3 5 7 9 11 13`

**2.2** ▷ `31 26 21 16 11 6 1`

**2.3** ▷ `3 5 9 17 33 65 129`

**2.4** ▷ `1 4 2 0 3 1 4 2 0 3 1 4 2 0 3 1 4`

## 2.2 Moyenne d'une série de valeurs

L'objectif est de calculer la moyenne d'une série de valeurs saisie par l'utilisateur, puis d'en afficher le résultat. L'utilisateur saisira les valeurs les unes après les autres, et terminera par la valeur particulière -1 pour signifier que la saisie est terminée.

**2.5** ▷ Faites le bilan des variables nécessaires dans l'algorithme et leur type,

**2.6** ▷ Écrivez l'algorithme en pseudo-code de calcul de la moyenne des valeurs saisies.

## 2.3 Une histoire de lapins transalpins

En 1202, un mathématicien italien connu sous le nom de Fibonacci a posé la question suivante : supposons qu'un couple (mâle–femelle) de lapins est né au début de l'année. Nous considérons que les conditions suivantes sont vérifiées :

— la maturité sexuelle du lapin est atteinte après un mois, qui est aussi la durée de gestation ;
— chaque portée comporte toujours un mâle et une femelle ;
— les lapins ne meurent pas.

Nous souhaitons connaître le nombre de couples de lapins au bout d'un an. Pour répondre à cette question, nous avons représenté figure 2.1 le nombre de couples pour les cinq premiers mois de l'année.

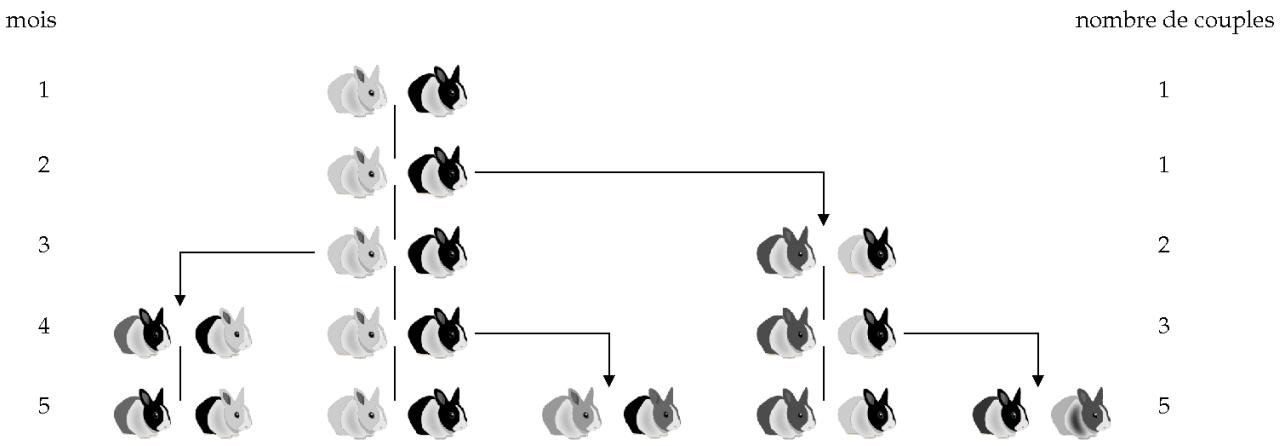



FIGURE 2.1 – Évolution du nombre de couples de lapins au cours des cinq premiers mois de l'année.

**2.7** ▷ Déterminez le nombre de couples de lapins au 6e mois, puis au 7e mois.

**2.8** ▷ En appelant $f_n$ le nombre de couples de lapins au ne mois, exprimez $f_n$ en fonction de $f_{n-1}$ et $f_{n-2}$. La suite $f_n$ est appelée *suite de Fibonacci*.

**2.9** ▷ Écrivez en pseudo-code l'algorithme qui calcule le nombre de couples de lapin au bout du ne mois, n étant choisi par l'utilisateur ($n > 2$). Ne cherchez pas à économiser les variables.

## 2.4 Intégration numérique

On cherche à calculer l'intégrale définie d'une fonction $f$ continue sur l'intervalle $[a, b]$. Si l'on connaît une primitive $F$ de $f$, on utilise alors la formule de Newton–Liebniz, qui exprime que l'intégrale est égale à la différence $F(b) - F(a)$ des valeurs prises par $F(x)$ aux extrémités de l'intervalle $[a, b]$. Si on ne connaît pas de primitive, on fait appel à des méthodes approchées, comme la méthode des rectangles ou la méthode des trapèzes. Vous étudierez ici ces deux dernières méthodes et écrirez les programmes qui les mettent en œuvre pour calculer l'intégrale d'une fonction.

### 2.4.1 Méthode des rectangles

Le procédé le plus simple de calcul approché d'une intégrale définie découle de la définition même de l'intégrale : on approche la fonction $f$ par une fonction en escalier.

Divisons l'intervalle $[a, b]$ en $n$ parties égales avec les points d'abscisse :

$$x_k = a + k\frac{(b-a)}{n} = a + kh, \quad k = 0, 1, \ldots, n-1$$

La méthode des rectangles consiste à décomposer l'aire comprise entre la courbe représentative de $f(x)$, l'axe des abscisses et les verticales $x = a$ et $x = b$. La décomposition est effectuée à l'aide de rectangles de largeur $(b-a)/n$ et dont l'un des sommets s'appuie sur la courbe. On obtient alors, selon que l'on se base sur la borne gauche ou la borne droite de chaque intervalle élémentaire pour construire chaque rectangle :

$$\int_a^b f(x)dx \approx \sum_{k=0}^{n-1}\left[f(x_k) \times \frac{(b-a)}{n}\right] \quad \text{ou} \quad \int_a^b f(x)dx \approx \sum_{k=1}^{n}\left[f(x_k) \times \frac{(b-a)}{n}\right]$$

Les données du problème sont les bornes $a$ et $b$ de l'intervalle d'intégration, et $n$ le nombre d'intervalles partiels dans l'intervalle $[a, b]$.

**2.10** ▷ Expérimentez graphiquement, sur du papier millimétré, les deux méthodes des rectangles pour la fonction $f(x) = x^2$, pour $n = 4$, pour $a = 0$ et $b = 1$. Déterminez alors les valeurs approchées de l'intégrale avec les deux méthodes.

**2.11** ▷ Calculez la valeur exacte de $\int_a^b f(x)dx$ et comparez avec les deux valeurs précédemment déterminées. Commentez vos résultats.

**2.12** ▷ Écrivez en pseudo-code l'algorithme permettant de calculer l'intégrale avec la méthode des rectangles (pour des rectangles dont le sommet gauche s'appuie sur la courbe).

### 2.4.2 Méthode des trapèzes

Il existe beaucoup d'autres méthodes d'intégration numérique dont l'une, la méthode des trapèzes, est une extension directe de la méthode des rectangles. Elle consiste à remplacer les rectangles par des trapèzes dont les deux sommets s'appuient sur la courbe de la fonction $f$.

**2.13** ▷ Montrez graphiquement que la surface du trapèze ainsi constitué est la moyenne des surfaces des rectangles obtenus avec les deux méthodes des rectangles.

**2.14** ▷ Déduisez-en que :

$$\int_a^b f(x)dx \approx \frac{1}{2}\left[f(a) \times \frac{b-a}{n} + 2f(x_1) \times \frac{b-a}{n} + \cdots + 2f(x_{n-1}) \times \frac{b-a}{n} + f(b) \times \frac{b-a}{n}\right]$$

**2.15** ▷ Écrivez en pseudo-code l'algorithme permettant de calculer l'intégrale de la fonction $f(x) = x^2$ en utilisant la méthode des trapèzes. L'utilisateur choisira les valeurs de $a$, $b$ et $n$.

**2.16** ▷ Optimisez l'algorithme pour accélérer la vitesse de traitement.

Fiche de TD n° 3

# Les listes et les fonctions

**Recommandations :** *Soyez rigoureux dans l'écriture des algorithmes en pseudo-code. Commencez par lister les variables utilisées en indiquant leur type et leur rôle si besoin. Définissez ensuite les fonctions s'il y en a, et enfin écrivez le contenu de l'algorithme. Indentez correctement votre pseudo-code pour en faciliter la lecture. Exécutez mentalement votre pseudo-code pour en vérifier le fonctionnement.*

## 3.1 Création de listes simples

Écrivez l'algorithme en pseudo-code permettant d'initialiser les trois listes `t1`, `t2`, `t3` suivantes :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 20 | 18 | 16 | 14 | 12 | 10 | 8 | 6 | 4 | 2 |

## 3.2 Recherche du maximum dans une liste

Écrivez un algorithme en pseudo-code capable de déterminer la valeur maximale d'une liste de nombres réels, ainsi que l'indice de ce maximum. La liste sera initialisée directement dans le pseudo-code. On affichera ensuite l'indice et la valeur du maximum.

## 3.3 Fonction de recherche du maximum dans une liste

Écrivez l'algorithme en pseudo-code de la fonction `max`, dont le rôle est de déterminer la valeur maximale d'une liste de nombres réels, ainsi que l'indice de ce maximum. La liste sera passée en paramètre de la fonction. La fonction renverra une liste de deux éléments, contenant le maximum et son indice.

## 3.4 Moyenne et écart-type expérimental d'une série de valeurs

L'objectif est de calculer la moyenne et l'écart-type d'une série de valeurs saisies par l'utilisateur, puis d'en afficher le résultat. L'utilisateur saisira les valeurs les unes après les autres, et terminera par la valeur particulière $-1$ pour signifier que la saisie est terminée. Les valeurs seront placées dans une liste.

**3.1** ▷ Faites le bilan des variables nécessaires dans l'algorithme et leur type.

**3.2** ▷ À l'aide d'une boucle, remplissez la liste à partir de ce que saisit l'utilisateur.

**3.3** ▷ À l'aide d'une deuxième boucle, déterminez la valeur moyenne des données contenues dans la liste.

**3.4** ▷ En réutilisant la valeur de la moyenne calculée précédemment, et à l'aide d'une boucle, calculez l'écart-type expérimental des données de la liste, défini par :

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$x_i$ désignant la i$^{e}$ valeur de la liste, et $\bar{x}$ la valeur moyenne des valeurs de cette liste.

# Algorithmique et Informatique

**M 1204**

1re année, IUT du Limousin

Département Mesures Physiques

— TRAVAUX PRATIQUES —

# Sommaire

# Première partie

# Apprentissage

# Recommandations

Les séances de travaux pratiques viennent compléter et illustrer concrètement l'enseignement théorique qui vous a été ou sera dispensé en cours et en travaux dirigés en 1re année.

L'objectif de cette première partie de la série de TP est de vous faire acquérir une compétence de base en représentation des données et en algorithmique. Cet apprentissage s'effectuera à l'aide du langage python, qui convient parfaitement pour débuter en programmation. C'est un langage moderne et de plus en plus utilisé. À titre d'exemple, il était utilisé par la NASA pour les missions de ses navettes, pour le développement de logiciels de contrôle aérien, ou encore en biologie moléculaire pour l'analyse des séquences d'ADN. Il paraît même qu'il est utilisé en 1re année de Mesures Physiques à Limoges !

Ces quatre premières séances sont d'une durée de 3 h chacune et sont effectuées individuellement. Quatre sujets vous sont proposés. Il est indispensable d'arriver à l'heure afin de pouvoir terminer le travail prévu. Aucun accès supplémentaire à la salle de TP ne sera autorisé pour terminer votre travail, sauf en cas de force majeure (dysfonctionnement du matériel ou du logiciel…).

— En cas d'absence justifiée, contactez le plus rapidement possible le responsable de la série de TP pour convenir avec lui d'une séance de rattrapage,
— En cas d'absence non justifiée, le TP ne sera pas rattrapé, et la note attribuée à cette séance sera égale à 0.

Afin de gérer correctement votre temps, chaque séance aura été **impérativement préparée à l'avance**. Lisez l'énoncé dans son intégralité. Ne cherchez pas à répondre tout de suite aux questions, mais essayez d'avoir une vue d'ensemble de ce que vous ferez pendant la séance de TP. Prenez connaissance des annexes, elles contiennent des notions et des informations pratiques nécessaires à une bonne compréhension du sujet.

**Le compte-rendu doit être rédigé dans votre cahier de laboratoire, qui sera rendu à l'enseignant en fin de séance**. Pour chaque sujet, une introduction présentera rapidement les objectifs de la séance. Répondez à toutes les questions qui vous sont posées dans l'énoncé. La présentation et l'orthographe seront pris en compte dans la notation. À la fin de la séance, imprimez vos programmes, et collez-les dans votre cahier. Veillez à ce qu'ils indiquent en commentaire en début de programme les éléments suivants :

— vos nom et prénom,
— votre numéro de groupe,
— le numéro du TP,
— la date,
— le nom du programme.

Sujet de TP 1

# Prise en main de Python et Wing IDE

Cette séance est une première approche de la programmation en Python, d'abord en mode interactif, puis en écrivant de petits programmes dans l'environnement de développement intégré Wing IDE. Vous étudierez d'abord quelques éléments de base du langage grâce à un programme founi. Ensuite, vous écrirez un petit jeu de golf. Vous serez guidés pas à pas dans le développement de ce programme.

Bonne découverte !

## 1.1 Python et le mode interactif

Il est temps de se mettre au travail. Plus exactement, nous allons demander à l'ordinateur de travailler à notre place, en lui donnant, par exemple, l'ordre d'effectuer une addition et d'afficher le résultat. Pour cela, nous allons devoir lui transmettre des « instructions », et également lui indiquer les « données » auxquelles nous voulons appliquer ces instructions.

### 1.1.1 Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Vous allez d'abord l'utiliser en mode interactif, c'est-à-dire d'une manière telle que vous pourrez dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les enregistrer sur disque.

Python peut être lancé depuis le menu `démarrer » Python 2.5 » Python (command line)`. Vous obtenez alors une fenêtre similaire à celle représentée sur la figure 1.1.

Les trois caractères « supérieur à » constituent le signal d'invite, ou prompt principal, lequel vous indique que Python est prêt à exécuter une commande. Par exemple, vous pouvez tout de suite utiliser python comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (ne recopiez pas le caractère # et le texte qui le suit, il s'agit de commentaires).

```
>>> 5+3          # le caractère # est utilisé pour les commentaires
>>> 2 - 9        # les espaces sont optionnels mais améliorent la lisibilité
>>> 7 + 3 * 4    # la hiérarchie conventionnelle des opérations
                 # mathématiques est-elle respectée ?
>>> (7+3)*4
>>> 20 / 3       # surprise !!!
```

Comme vous pouvez le constater, les opérateurs arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement `+`, `-`, `*` et `/`. Les parenthèses sont fonctionnelles.

Par défaut, la division est cependant une division euclidienne, ce qui signifie que si on lui fournit des arguments qui sont des nombres entiers, le résultat de la division est lui-même un entier, comme

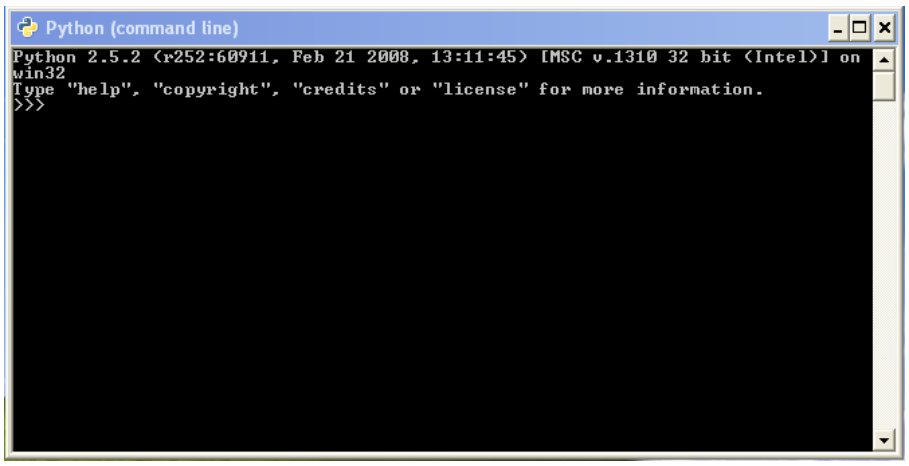



FIGURE 1.1 – Python en mode interactif.

dans le dernier exemple ci-dessus [1]. Si vous voulez qu'un argument soit compris par Python comme étant un nombre réel, il faut le préciser, en fournissant au moins un point décimal [2]. Essayez par exemple :

```
>>> 20.0 / 3        # (comparez le résultat avec celui obtenu
>>> 8./5            # à l'exercice précédent)
>>> float(20) / 3   # conversion en réel d'un des deux opérandes
```

Si une opération est effectuée avec des arguments de types mélangés (entiers et réels), Python convertit automatiquement les opérandes en réels avant d'effectuer l'opération. Essayez :

```
>>> 4 * 2.5 / 3.3
```

### 1.1.2 Données et variables

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses, mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de nombres binaires.

Pour pouvoir accéder aux données, le programme d'ordinateur fait abondamment usage d'un grand nombre de variables de différents types (le type « entier », le type « réel », le type « chaîne de caractères »…).

Pour choisir le nom de vos variables, consultez l'annexe B page .

### 1.1.3 Affecter une valeur à une variable

Les termes « affecter une valeur », « assigner une valeur » ou « donner une valeur » à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe = (*égal*) :

```
>>> n = 7                      # donner à n la valeur 7
>>> msg = "Quoi de neuf ?"     # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159               # assigner sa valeur à la variable pi
```

1. À partir de Python 3, la division n'est plus euclidienne par défaut, et 1/2 donne bien 0.5
2. Dans tous les langages de programmation, les conventions mathématiques de base sont celles en vigueur dans les pays anglophones : le séparateur décimal sera donc toujours un point, et non une virgule comme chez nous. Dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres « à virgule flottante », ou encore des nombres « de type float »

Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

— trois noms de variables, à savoir `n`, `msg` et `pi`
— trois séquences d'octets, où sont encodées le nombre entier 7, la chaîne de caractères `Quoi de neuf ?` et le nombre réel 3,14159.

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

— créer une variable ;
— lui attribuer un type bien déterminé ;
— lui attribuer une valeur particulière ;

**Important**

Ce qu'on place à gauche du signe égal doit toujours être une variable et non une expression (combinaison de variables et d'opérateurs).

Par exemple, `m + 1 = b` est *illégal* car `m + 1` n'est pas un nom de variable.

En revanche, écrire `a = a + 1` est inacceptable en mathématique, alors que cette forme d'écriture est très fréquente en programmation. Cette instruction signifie en l'occurence « augmenter la valeur de la variable `a` d'une unité », ou « incrémenter `a` ».

### 1.1.4 Affichage de la valeur d'une variable

Pour afficher la valeur d'une variable à l'écran, il existe deux possibilités. La première consiste à saisir au clavier le nom de la variable, puis à appuyer sur la touche `Entrée`. Python répond en affichant la valeur correspondante :

```
>>> n
7
>>> msg
"Quoi de neuf ?"
>>> pi
3.14159
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours l'instruction `print` :

```
>>> print msg
Quoi de neuf ?
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. L'instruction print n'affiche strictement que la valeur de la variable, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des guillemets (afin de vous rappeler le type de la variable).

## 1.2 Écrire des programmes : Wing IDE

Wing IDE (Integrated Developement Environment ou environnement de développement intégré) est un logiciel vous permettant d'écrire des programmes en Python, de les déboguer, et de les exécuter tout en restant dans la même application. Il a été écrit par les concepteurs même de Python, et sa version éducation est téléchargeable gratuitement sur internet (disponible également sur Claroline).

Avant toute chose, vous allez d'abord organiser votre espace disque personnel. À la racine de votre espace personnel (lecteur `P`), créez un dossier nommé `TP_INFO_MP1`. Allez ensuite dans ce dossier. Cette première partie comportant quatre séances de travaux pratiques, vous allez créer un

dossier par séance. Créez un dossier nommé `TP1`. Tous les programmes que vous allez écrire aujourd'hui seront stockés dans ce dossier. Créez également les dossiers `TP2` à `TP4` pour les séances futures.

### 1.2.1 Description de l'environnement de développement

Pour lancer Wing IDE, cliquez sur `Démarrer » Programmes » Wing IDE 101`. Acceptez ensuite la licence d'utilisation. L'interface de Wing IDE s'affiche ensuite (figure 1.2).

#### Configuration de Wing IDE

Allez ensuite dans le menu `Edit » Preferences`. Vous allez d'abord configurer la langue de l'interface. Dans la liste déroulante `Display language`, choisissez `Français`. Vous allez maintenant configurer le dossier de travail par défaut. Cliquez ensuite sur `Files`. Dans la liste déroulante `Default directory policy`, choisissez `Use Fixed Directory Specified Below`, cliquez ensuite sur le bouton `Browse` et choisissez le dossier `P:\TP_INFO_MP1`. Validez par `Ok`, puis cliquez sur `Restart Now`. C'est prêt !

#### Éléments de l'interface

Nous allons décrire brièvement les différents éléments qui composent cette interface. Il est possible que l'interface se présente de manière légèrement différente car les éléments sont repositionnables.

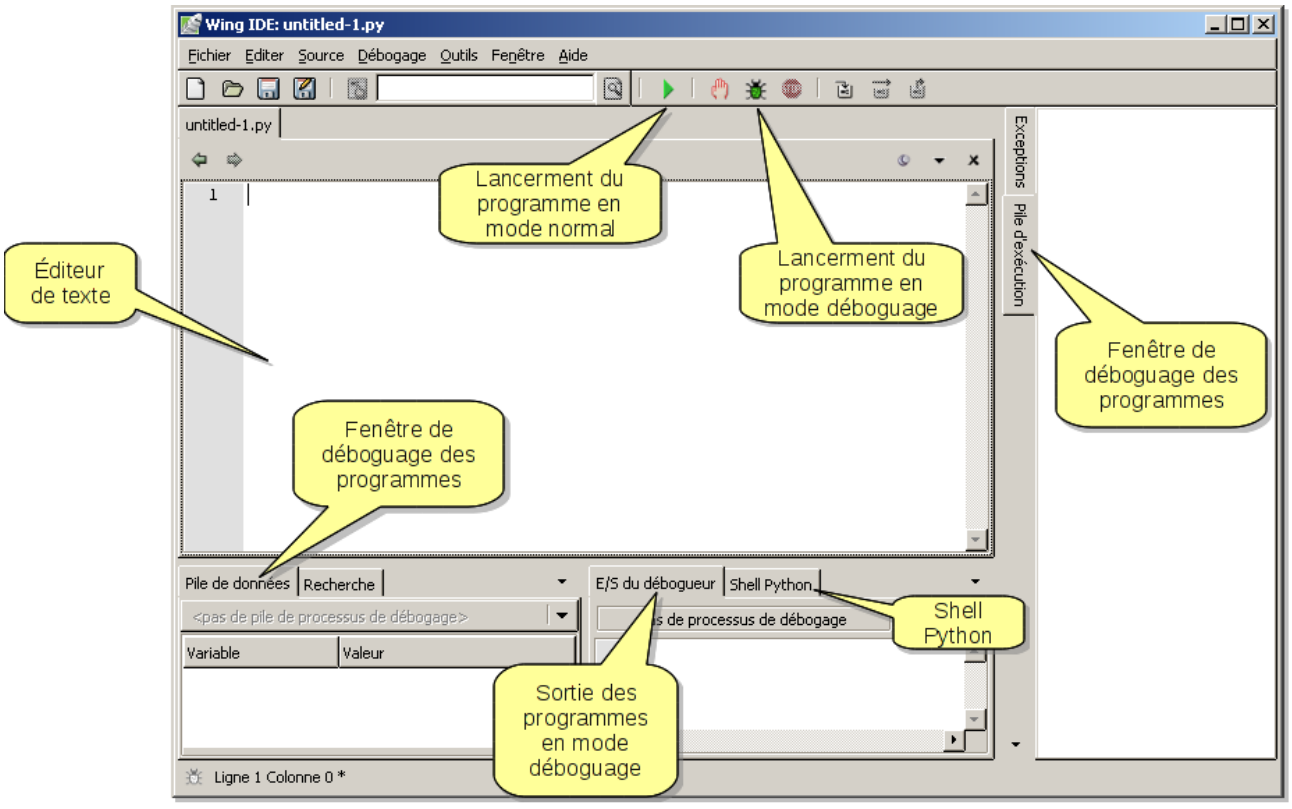



FIGURE 1.2 – Interface de Wing IDE.

**Éditeur de texte :** il s'agit du cœur de l'interface. C'est ici que vous saisirez vos programmes, c'est-à-dire la suite des instructions Python que vous souhaitez faire exécuter par la machine. Une fois que vous avez saisi votre programme, il faut l'enregistrer sur le disque dur avant

de pouvoir l'exécuter. Chaque programme que vous écrirez correspondra à un fichier unique portant l'extension `.py` ; vous veillerez à utiliser des noms de fichiers qui soient en relation directe avec ce que fait le programme [3] ;

**Shell Python :** cet onglet correspond exactement à ce que vous avez vu dans la première partie du TP. Il s'agit d'une console Python en mode texte dans laquelle s'exécute un interpréteur Python. Vous pouvez y saisir des lignes de commandes pour tester quelques lignes afin d'analyser le résultat. C'est également ici que s'exécute le programme que vous écrivez dans l'éditeur lorsque vous le lancez en mode normal ( ▶ ) ;

**E/S du débogueur :** cet onglet affiche la sortie de votre programme en mode déboguage (mode permettant d'analyser pas à pas ce que fait le programme afin de détecter ses éventuelles erreurs de conception). C'est également ici que s'exécute le programme que vous écrivez dans l'éditeur lorsque vous le lancez en mode déboguage ( 🐞 ou raccourci clavier F5) ;

▶ **:** utilisez ce bouton pour exécuter votre programme en mode normal, dans la console Python. Il est impossible alors de travailler en mode pas à pas pour analyser son déroulement ;

🐞 **:** utilisez ce bouton pour exécuter votre programme en mode déboguage. Vous pouvez alors placer des points d'arrêt (le programme se met en pause et vous pouvez regarder ce qu'il se passe), exécuter en mode pas à pas (ligne par ligne)... **Lors des séances de TP, vous utiliserez cette méthode pour exécuter les programmes** ;

**Pile de données et pile d'exécution :** en mode déboguage, ces deux onglets vous permettent d'ausculter le programme : ligne du programme en cours, erreurs détectées, valeurs des variables utilisées par le programme... Nous reviendrons plus en détail sur le déboguage un peu plus loin.

### 1.2.2 Votre premier programme

Cette fois-ci, on entre dans le cœur du sujet. Vous allez saisir puis exécuter votre premier programme en Python en utilisant WingIDE !

Pour commencer un nouveau programme, cliquez sur... `Nouveau` ou utilisez le raccourci-clavier `Ctrl + N`. Une page blanche apparaît alors dans l'éditeur de texte. Saisissez le listing ci-après. Ne vous inquiétez pas si vous ne comprenez pas tout ce que vous saisissez [4], nous reprendrons un peu plus loin les éléments nécessaires du langage Python.

Listing 1.1 – Votre premier programme

```
1  # coding:latin1
2  # programme de conversion degrés Celsius <-> degrés Fahrenheit
3  # [INSÉREZ ICI VOTRE NOM]
4  # TP1, le [INSÉREZ ICI LA DATE]
5  # conversion.py
6
7  # Variables :
8  # reponse : entier
9  # Tc : réel, température en degrés Celsius
10 # Tf : réel, température en degrés Fahrenheit
11
12 print u"programme de conversion degrés Celsius <-> degrés Fahrenheit"
13 print u"1. Convertir de degrés Celsius vers degrés Fahrenheit"
14 print u"2. Convertir de degrés Fahrenheit vers degrés Celsius"
15 print
16 reponse = input("Votre choix : ")
17
```

3. Pour les noms de fichiers, n'utilisez que les lettres minuscules non accentuées, ainsi que le caractère de soulignement _ (souligné). N'utilisez pas d'autres caractères, en particulier les accents, les espaces, les antislashs (\)... Ces recommandations sont valables quel que soit le langage de programmation.

4. jeux de mots !

```
18 if reponse == 1:
19     # conversion degrés Celsius->degrés Fahrenheit
20     print u"Saisissez la température en °C"
21     Tc = input("Tc = ")
22     Tf = (9./5) * Tc + 32
23     print "Tf = " , Tf , "°F"
24 elif reponse == 2:
25     # conversion degrés Fahrenheit->degrés Celsius
26     print u"Saisissez la température en °F"
27     Tf = input("Tf = ")
28     Tc = (Tf - 32) / (9./5)
29     print "Tc = " , Tc , "°C"
30 else:
31     # autre réponse
32     print "choix non valable"
```

Ce programme permet de convertir une température exprimée en degrés Celsius vers des degrés Fahrenheit, et inversement, suivant la relation suivante :

$$T_F = (9/5) * T_c + 32 \tag{1.1}$$

Le sens de la conversion est choisi par l'utilisateur par le biais d'un petit menu proposant deux options. En fonction de la réponse (`1` ou `2`), les lignes de code adéquates sont exécutées.

Enregistrez ce programme dans le dossier `TP1` sous le nom `conversions.py`. Appuyez sur `F5` pour exécuter le programme.

Nous allons maintenant analyser brièvement les éléments de langage qui se trouvent dans ce programme.

#### 1.2.2.1 Encodage des caractères

La ligne 1 permet de spécifier l'encodage des caractères. Vous indiquez au programme que vous utiliserez un jeu de caractères des langues occidentales (latin1 pour le français). Vous pouvez alors utiliser les lettres accentuées dans vos programmes (en particulier dans les chaînes de caractères et dans les commentaires).

#### 1.2.2.2 Les commentaires

Les commentaires commencent par le caractère dièse (#). Tout le texte qui suit jusqu'à la fin de la ligne est ignoré au moment de l'exécution du programme. Le commentaire n'est donc là que pour aider à la compréhension du programme. L'utilisateur qui exécute le programme ne voit pas les commentaires, à moins bien sûr qu'il jette un œil au code source du programme ! Notez que le caractère # est également utilisé pour préciser l'encodage des caractères (ligne 1).

#### 1.2.2.3 Écrire à l'écran

La fonction `print` permet d'écrire du texte à l'écran. Le texte, ou chaîne de caractères, doit se trouver entre des guillemets simples (`'`) ou doubles (`"`). Vous pouvez également afficher la valeur d'une variable en utilisant `print` suivi du nom de la variable. Si vous souhaitez afficher plusieurs éléments sur la même ligne, séparez-les par des virgules. Enfin, après l'exécution de `print`, le curseur est ramené en début de la ligne suivante. Notez la présence du caractère `u` juste avant la chaîne de caractères. Il permet un affichage correct des accents dans le Shell Python et dans l'onglet `E/S du débogueur`. Si la chaîne à afficher ne contient pas d'accents, vous pouvez éventuellement vous en passer.

#### 1.2.2.4 Lire des valeurs

Il n'est pas rare que vous ayez à demander des informations à l'utilisateur, que ce soit du texte ou des nombres. Pour lire[5] une valeur, utilisez la syntaxe de la ligne 16. Avec la commande `input`, le programme affiche d'abord le texte à l'écran (ici `Votre choix :` ), puis attend ensuite que l'utilisateur saisisse une valeur au clavier et valide avec la touche `Entrée`. La valeur saisie est alors stockée dans la variable qui se trouve à gauche du signe égal (ici `reponse`).

#### 1.2.2.5 Instructions conditionnelles

L'instruction `if` (ligne 18) permet de tester si une expression est vraie ou fausse. Si elle est vraie, les lignes **indentées**[6] suivantes sont exécutées. Si elle est fausse, ces lignes ne sont pas exécutées et on continue à la première ligne suivante non indentée.

Si on souhaite exécuter d'autres instructions si et seulement si l'expression est fausse, on utilisera `else` qui signifie sinon. Encore une fois il faudra prendre bien soin d'indenter les lignes concernées.

Enfin on peut utiliser le mot clé `elif` (sinon si) qui permet, si la première expression est fausse, de refaire un nouveau test sur une autre expression.

Notez que l'opérateur de **comparaison** est le signe double égal ==, qu'il ne faut pas confondre avec l'opérateur d'affectation (=). Mais ne vous inquiétez pas, si vous mettez = à la place de ==, Python va râler au moment de l'exécution. Faites le test et notez l'erreur renvoyée par Python.

N'oubliez pas le signe `:` à la fin des lignes commençant par `if`, `elif` ou `else`, et qui marque le début d'un bloc d'instructions qui seront indentées.

**1.1** ▷ Modifiez le programme pour que les unités °C et °F s'affichent correctement.

## 1.3 Une petite partie de golf

L'objectif est ici d'écrire un petit jeu qui simule une partie de golf. L'ordinateur choisit la distance séparant le joueur du trou, et le but du jeu est d'envoyer la balle le plus près du trou. Pour cela, le joueur choisit l'angle et la vitesse initiaux de la balle. Le programme calcule alors la distance à laquelle la balle touche le sol à partir des équations régissant sa trajectoire. Afin de simplifier le problème, quelques approximations seront faites :

— les effets de la présence d'une atmosphère (frottements, vents…) ne sont pas pris en compte,
— la balle ne rebondit pas et ne roule pas lorsqu'elle touche le sol.

Vous commencerez par écrire un programme très simple, puis au fil des sections, vous ajouterez progressivement de nouvelles fonctionnalités. À chaque étape, vous ferez constater à l'enseignant le bon fonctionnement de votre programme. Vous ne passerez à l'étape suivante que lorsque vous aurez le feu vert de l'enseignant.

### 1.3.1 Programme de base

**1.2** ▷ En vous aidant du pseudo-code du programme de base vu en travaux dirigés, écrivez le programme en langage Python. Votre programme sera enregistré dans le dossier `TP1` sous le nom `golf.py`. Au début de ce programme, vous placerez en commentaires vos nom, prénom, groupe de TP, la date et le nom du programme. N'oubliez pas de commenter votre programme.

**Quelques remarques :**

— Pour calculer la racine carrée d'un nombre, utilisez la fonction `sqrt()`,
— mais vous devez pour cela ajouter `from math import sqrt` au début de votre programme,

5. Lorsque on dit à la machine de lire une valeur, cela implique que l'utilisateur va devoir écrire cette valeur. Et quand on demande à la machine d'écrire une valeur, c'est pour que l'utilisateur puisse la lire. Lecture et écriture sont donc des termes qui, comme toujours en programmation, doivent être compris du point de vue de la machine.

6. Une ligne est indentée si on place au début de celle-ci une tabulation qui va la décaler vers la droite d'un certain nombre de caractères.

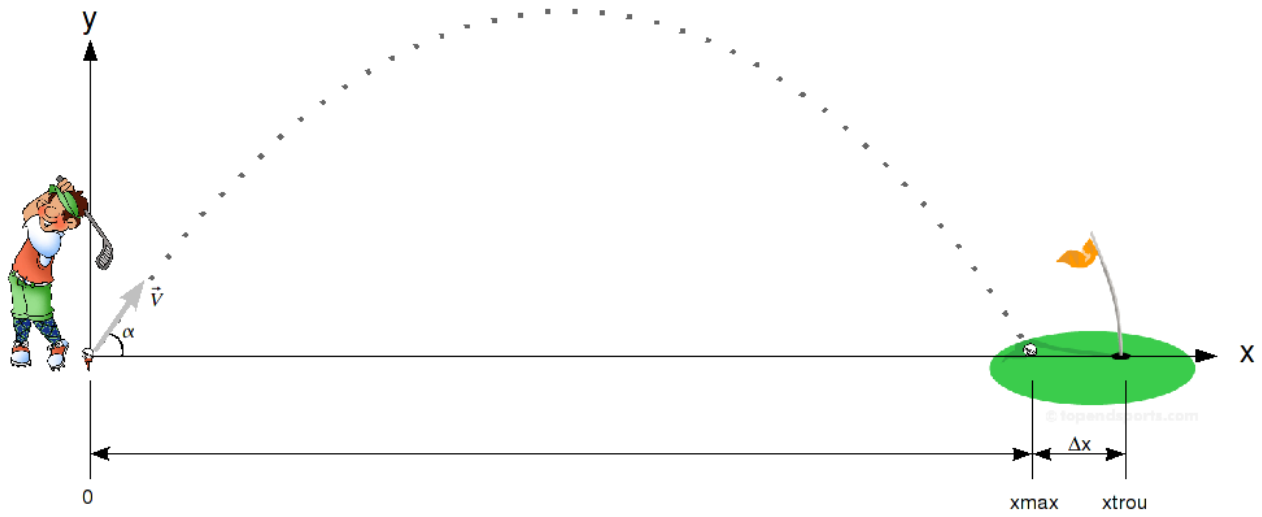



FIGURE 1.3 – Une petite partie de golf.

— vous devrez également importer les fonctions `cos` et `sin` depuis le module `math` : `from math import sqrt, cos, sin`

La documentation des fonctions du module `math` est disponible à l'adresse :
`https://docs.python.org/2.7/library/math.html`

**1.3** ▷ Cherchez dans cette documentation comment utiliser la valeur de $\pi$. Utilisez pour cela l'entrée `constants` de la table des matières de la documentation.

### 1.3.2 Placement du trou

Le jeu est pour l'instant sans grand intérêt car on ne sait pas à quelle distance se situe le trou. Pour remédier à cela, l'ordinateur choisira aléatoirement cette distance et la placera dans la variable `xtrou`.

Pour tirer au sort un nombre entier entre `a` et `b`, vous devez :

— placer `from random import randint` au début de votre programme,

— utiliser la fonction `randint(a,b)` qui vous renvoie un nombre entier au hasard (ou presque !) compris entre $a$ et $b$.

La documentation de la fonction `randint` est disponible à l'adresse :
`https://docs.python.org/2.7/library/random.html`

**1.4** ▷ Testez la fonction `randint()` dans le Shell Python pour mieux voir ce que ça donne (n'oubliez pas de saisir `from random import randint` **avant**). Expliquez le fonctionnement de cette fonction dans votre compte-rendu.

**1.5** ▷ Placez dans la variable `xtrou` un nombre aléatoire compris entre 100 et 300 (inclus).

**1.6** ▷ Modifiez le programme pour afficher à l'écran la position du trou afin que l'utilisateur choisisse au mieux l'angle et la vitesse initiaux.

**1.7** ▷ Améliorez l'affichage de la valeur de `xmax` en n'affichant à l'écran qu'un chiffre après la virgule. Dans la ligne du programme qui réalise cet affichage, remplacez `xmax` par `round(xmax,1)`. Vous l'aurez compris, le premier argument correspond au nom de la variable, et le second au nombre de chiffres après la virgule à afficher[7]. Testez la fonction `round()` dans le shell Python, et expliquez son fonctionnement dans votre compte-rendu.

La documentation de la fonction `round` est disponible à l'adresse :
`https://docs.python.org/2.7/library/functions.html`

7. Notez bien que cette fonction n'affecte que l'affichage de la variable à l'écran, et ne modifie pas la valeur de celle-ci.

### 1.3.3 Ajout du green

Vous vous êtes probablement rendu(e) compte qu'il est très difficile d'envoyer la balle directement dans le trou ! Nous nous contenterons de mettre la balle sur le green.

**1.8** ▷ Définissez la variable `rayonGreen` et placez-y un nombre entier tiré au sort entre 10 et 20,

**1.9** ▷ Informez le joueur de ce rayon (après avoir affiché la distance à laquelle se trouve le trou),

**1.10** ▷ Après avoir affiché `xmax`, vous calculerez la distance `deltax` entre la balle et le trou, et l'afficherez à l'écran avec un chiffre après la virgule. Vous aurez besoin de la fonction `abs()` qui vous renvoie la valeur absolue de l'expression que vous lui passez en paramètre.

**1.11** ▷ Vous informerez ensuite le joueur s'il est ou non sur le green.

Dans sa version finale, le programme affichera exactement ce qui suit (les valeurs numériques peuvent être différentes) :

```
Le trou se trouve à 209 m
Rayon du green : 15 m
angle initial (degré) : 45
vitesse initiale (km/h) : 170
xmax = 227.5 m
vous êtes à 18.6 m du trou
Pas sur le green
```

**1.12** ▷ Faites constater le bon fonctionnement de votre programme à l'enseignant.

## Bonus

**1.13** ▷ Traitez le cas où la balle se trouve initialement à une altitude $y_0$ tirée au sort entre 0 et $5\ \mathrm{m}$, et le green à une altitude $y_g$ tirée au sort entre 0 et $5\ \mathrm{m}$.

**1.14** ▷ Le discriminant du polynôme peut maintenant être négatif. Quelle est alors la signification physique de cette situation particulière ?

**1.15** ▷ En utilisant la fonction `exit()` du module `sys`, modifiez le programme pour quitter si le discriminant est négatif. Un message d'alerte sera affiché à l'utilisateur.

La documentation de la fonction `exit` est disponible à l'adresse :

`http://www.python.org/doc/2.7/library/sys.html`

Sujet de TP 2

# Utilisation des boucles

Vous apprendrez dans ce TP à manipuler les boucles (instructions de répétition), et en particulier la boucle `while` (tant que). Vous appliquerez cette notion au calcul de la moyenne et de l'écart-type d'une série de données. Dans un deuxième temps, vous étudierez la suite de Fibonacci en déterminant l'évolution au cours du temps d'une population de lapins.

## 2.1 Utilisation des boucles

### 2.1.1 La boucle `while`

Les boucles (instructions de répétition) permettent de faire exécuter plusieurs fois certaines parties d'un programme. Vous allez travailler ici avec la boucle `while`, qui permet de répéter une séquence d'instructions tant qu'une condition est vraie. La syntaxe de la boucle `while` est la suivante :

Listing 2.1 – Syntaxe de la boucle while

```
1 while <condition>:
2     <instruction_1>
3     <instruction_2>
4 <instruction_3>
```

et en pseudo-code :

```
TANT_QUE (condition) FAIRE
    | instruction_1
    | instruction_2
FIN_TANT_QUE
instruction_3
```

Le programme commence par tester la condition. Si elle est vraie, les instructions 1 et 2 sont exécutées, puis la condition est testée de nouveau. Si cette condition est toujours vraie, on exécute de nouveau les instructions 1 et 2, etc. Si elle est fausse, on passe directement à l'instruction 3 et le programme poursuit son déroulement.

La répétition est très souvent contrôlée par un compteur. Dans ce cas, elle exige :

— le nom d'une variable de contrôle (ou compteur de boucle) ;
— une valeur initiale de la variable de contrôle ;
— la condition qui vérifie si la valeur finale de la variable de contrôle est atteinte ;
— l'incrémentation (ou décrémentation) de la variable de contrôle qui modifie sa valeur à chaque passage dans la boucle.

Voici un exemple très simple où l'on affiche les nombres de 1 à 10. Testez-le dans le Shell Python (n'oubliez pas les deux points et les tabulations).

L'initialisation de la variable de contrôle est réalisée par `compteur = 1`. Les instructions indentées après le `while` constituent le *corps* de la boucle.

L'instruction `compteur = compteur + 1` incrémente (c'est-à-dire ajoute 1) à la variable de contrôle. En général, cette instruction doit être la dernière du corps de la boucle.

Listing 2.2 – Affichage des nombres de 1 à 10

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

La condition de répétition de la boucle permet de tester si la valeur de la variable de contrôle est inférieure ou égale à 10. Dans l'exemple, 10 est la dernière valeur pour laquelle la condition est vraie. La boucle se termine lorsque la valeur de la variable de contrôle dépasse 10, c'est-à-dire lorsque `compteur` a la valeur 11. Vérifiez la valeur de la variable compteur.

Il peut arriver que le corps d'une boucle `while` ne s'exécute pas. En effet, la condition est vérifiée avant tout traitement et peut être fausse dès le début.

### 2.1.2 Un peu de statistiques descriptives

On a mesuré de manière répétitive la tension du réseau électrique à l'aide d'un multimètre. On a relevé les 20 indications données dans le tableau 4.1.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 234,7 | 234,9 | 234,8 | 234,6 | 234,7 | 235,0 | 235,0 | 234,7 | 234,8 | 234,8 |
| 234,7 | 235,0 | 234,8 | 234,7 | 234,9 | 234,9 | 234,9 | 235,0 | 234,8 | 234,9 |

TABLE 2.1 – Indications mesurées ($l_v/V$)

#### 2.1.2.1 Calcul de la moyenne

L'objectif est d'écrire un programme qui calcule la moyenne des valeurs de ce tableau en utilisant une boucle `while`. Les tensions seront placées en début de programme dans une liste[1] nommée `tension` en utilisant la syntaxe suivante :

```
tension = [ 234.7 , 234.9 , 234.8 , ...]
```

Notez que le séparateur décimal est le point, et que les éléments de la liste sont séparés par des virgules. Chaque valeur de la liste est associée à un indice allant de 0 à 19. Pour accéder à un élément de la liste, vous utiliserez la syntaxe suivante :

```
>>> print tension[1]
234.9
```

De manière plus générale, vous pouvez accéder successivement à tous les éléments de la liste `tension` en utilisant un compteur `i` que vous ferez évoluer de 0 à 19.

**2.1** ▷ En vous aidant de l'algorithme en pseudo-code étudié en travaux dirigés, écrivez le programme de calcul de la moyenne de la série de données et enregistrerez-le dans le dossier `TP2` sous le nom `moyenne.py`. Vous déterminerez la dimension de la liste `tension` en utilisant la fonction `len()` :

```
>>> len(tension)
20
```

Vous afficherez la valeur de la moyenne avec deux chiffres après la virgule.

1. Les listes sont très similaires aux tableaux que l'on rencontre dans d'autres langages de programmation. Une liste peut cependant contenir des éléments de types différents, contrairement aux tableaux.

#### 2.1.2.2 Calcul de l'écart-type

**2.2** ▷ Complétez dans votre compte-rendu l'algorithme de calcul de la moyenne pour calculer également l'écart-type expérimental défini par :

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$x_i$ étant le résultat du i$^{e}$ mesurage et $\bar{x}$ la moyenne arithmétique des résultats considérés.

La fonction puissance est obtenue avec l'opérateur `**`. Exemple :

```
>>> 2**3
8
```

Faites vérifier l'algorithme par l'enseignant avant de passer à la suite.

**2.3** ▷ Enregistrez le programme précédent dans le dossier `TP2` sous le nom `ecart_type.py`, et complétez-le pour calculer l'écart-type expérimental. Vous afficherez la valeur de l'écart-type avec trois chiffres significatifs.

### 2.1.3 La boucle `for ... in ...`

Cette structure permet également de créer une boucle pour itérer sur tous les éléments d'une liste. Voici un exemple d'utilisation de cette structure de contrôle :

```
1 valeurs = [12.3 , 8,6 , 6.7]
2 for valeur in valeurs:
3   print valeur,
```

Ce programme affichera

```
12.3 8.6 6.7
```

**2.4** ▷ complétez le programme `ecart_type.py` en ajoutant à nouveau le calcul de l'écart-type, mais en utilisant la structure de contrôle `for ... in ...`

## 2.2 Une histoire de lapins transalpins

En 1202, un mathématicien italien connu sous le nom de Fibonacci a posé la question suivante : supposons qu'un couple (mâle–femelle) de lapins est né au début de l'année. Nous considérons que les conditions suivantes sont vérifiées :

— la maturité sexuelle du lapin est atteinte après un mois, qui est aussi la durée de gestation ;
— chaque portée comporte toujours un mâle et une femelle ;
— les lapins ne meurent pas.

Nous souhaitons connaître le nombre de couples de lapins au bout d'un nombre quelconque de mois.

**2.5** ▷ En vous aidant de l'algorithme en pseudo-code vu en travaux dirigés, écrivez le programme déterminant le nombre de couples de lapin au mois $n$ ($n > 2$) et enregistrez-le sous le nom `fibonacci.py`.

## Bonus

**2.6** ▷ Modifiez le programme pour afficher le nombre de couples pour chaque mois, ainsi que le rapport entre le nombre de couples du mois et le nombre de couples du mois précédent. Vérifiez que ce rapport tend vers le nombre d'or défini par :

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1,618033988$$

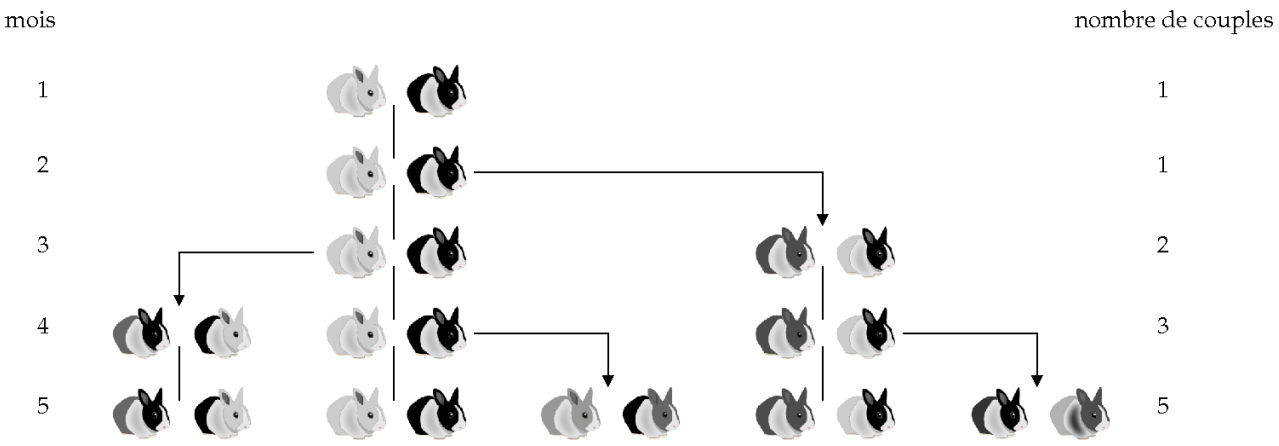



FIGURE 2.1 – Évolution du nombre de couples de lapins au cours des cinq premiers mois de l'année.

Votre programme devra afficher exactement :

```
nombre de mois : 10
f 2 = 2   rapport = 2.0
f 3 = 3   rapport = 1.5
f 4 = 5   rapport = 1.66666666667
f 5 = 8   rapport = 1.6
f 6 = 13   rapport = 1.625
f 7 = 21   rapport = 1.61538461538
f 8 = 34   rapport = 1.61904761905
f 9 = 55   rapport = 1.61764705882
```

**2.7** ▷ En vous aidant de l'annexe C, formatez l'affichage de façon à obtenir le résultat suivant à l'écran :

```
nombre de mois : 10
f2 = 2        r = 2.00
f3 = 3        r = 1.50
f4 = 5        r = 1.67
...
```

Sujet de TP 3

# Intégration numérique

Ce TP concerne l'étude de l'intégration numérique d'une fonction par la méthode des rectangles, puis par la méthode des trapèzes. Vous étudierez ces deux méthodes, et comparerez leurs exactitudes.

On cherche à calculer l'intégrale définie d'une fonction $f$ continue sur l'intervalle $[a, b]$. Si l'on connaît une primitive $F$ de $f$, on utilise alors la formule de Newton–Liebniz, qui exprime que l'intégrale est égale à la différence $F(b) - F(a)$ des valeurs prises par $F(x)$ aux extrémités de l'intervalle $[a, b]$. Si on ne connaît pas de primitive, on fait appel à des méthodes approchées, comme la méthode des rectangles ou la méthode des trapèzes. Vous étudierez ici ces deux dernières méthodes et écrirez les programmes qui les mettent en œuvre pour calculer l'intégrale d'une fonction.

Avant toute chose, vous allez apprendre à utiliser l'éditeur Wing-IDE en mode débogage. Ce mode vous sera particulièrement utile dans cette séance de TP pour comprendre comment l'intégration numérique fonctionne.

## 3.1 Débogage : points d'arrêt et exécution pas à pas

Lorsqu'un script contient des bogues, il est intéressant de pouvoir y placer des poses afin de consulter la valeur des différentes variables, et d'exécuter ensuite le script ligne par ligne pour constater l'évolution du contenu des différentes variables. L'éditeur Wing-IDE permet ceci, à partir du moment où le script est lancé en mode débogage (touche F5).

Pour permettre au script de se mettre en pause à l'endroit voulu, il faut placer des points d'arrêts :

— soit en cliquant entre le numéro de la ligne et la colonne,

— soit en appuyant sur la touche F9

Il suffit ensuite de lancer le programme en mode débogage. Lorsqu'un point d'arrêt est rencontré, la ligne concernée est placée sur fond rouge (notez qu'elle n'est pas encore exécutée). L'onglet « `Pile de données` » permet d'afficher la valeur de chaque variable du script. L'onglet « `E/S du débogueur` » affiche lui la sortie du programme.

— pour relancer le programme jusqu'à ce qu'il rencontre un autre point d'arrêt ou qu'il se termine, appuyez de nouveau sur F5

— pour exécuter les lignes suivantes les unes après les autres, appuyez sur F6

— pour arrêter le programme et quitter le mode débogage, appuyez sur CTRL + F5

**3.1** ▷ testez le débogage dans Wing-IDE avec le petit programme donné sur la capture d'écran de la figure 3.1.

## 3.2 Méthode des rectangles

Le procédé le plus simple de calcul approché d'une intégrale définie découle de la définition même de l'intégrale : on approche la fonction $f$ par une fonction en escalier.

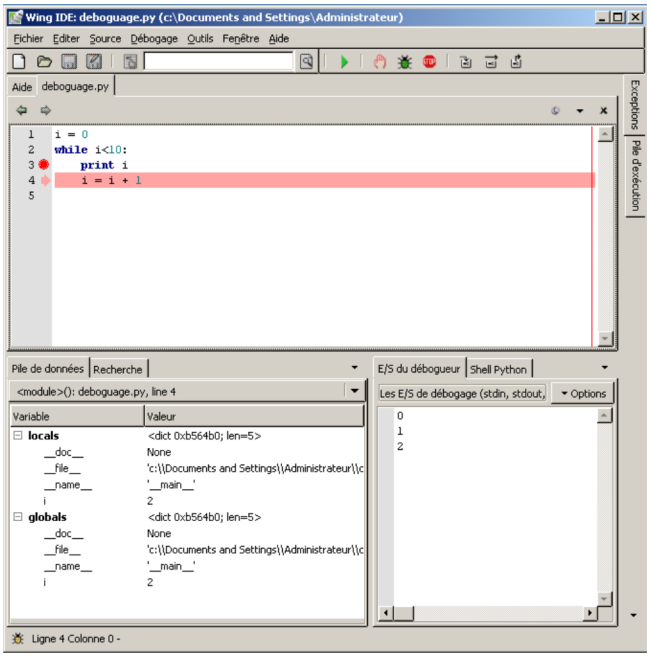



FIGURE 3.1 – Wing-IDE en mode débogage.

Divisons l'intervalle $[a, b]$ en $n$ parties égales avec les points d'abscisse :

$$x_k = a + k\frac{(b-a)}{n} = a + kh, \quad k = 0, 1, \ldots, n-1$$

La méthode des rectangles consiste à décomposer l'aire comprise entre la courbe représentative de $f(x)$, l'axe des abscisses et les verticales $x = a$ et $x = b$. La décomposition est effectuée à l'aide de rectangles de largeur $(b-a)/n$ et dont l'un des sommets s'appuie sur la courbe. On obtient alors, selon que l'on se base sur la borne gauche ou la borne droite de chaque intervalle élémentaire pour construire chaque rectangle :

$$\int_a^b f(x)dx \approx \sum_{k=0}^{n-1}\left[f(x_k) \times \frac{(b-a)}{n}\right] \quad \text{ou} \quad \int_a^b f(x)dx \approx \sum_{k=1}^{n}\left[f(x_k) \times \frac{(b-a)}{n}\right] \tag{3.1}$$

Les données du problème sont les bornes $a$ et $b$ de l'intervalle d'intégration, et $n$ le nombre d'intervalles partiels dans l'intervalle $[a, b]$.

**3.2** ▷ En vous aidant de l'algorithme en pseudo-code étudié en travaux dirigés, écrivez le programme permettant de calculer l'intégrale `integr_gauche` de la fonction $f(x) = x^2$ pour des rectangles dont le sommet gauche s'appuie sur la courbe, et `integr_droite` pour des rectangles dont le sommet droit s'appuie sur la courbe. L'utilisateur choisira les valeurs de $a$, $b$ et $n$. La valeur de l'intégrale sera affichée avec cinq chiffres après la virgule. Vous enregistrerez le programme dans le dossier `TP3` sous le nom `rectangles.py`.

**3.3** ▷ Optimisez le code de votre programme pour accélérer la vitesse de traitement,

**3.4** ▷ En utilisant ce programme, remplissez le tableau 3.1 et commentez-le.

| n | 1 | 2 | 4 | 50 | 100 | 1 000 | 10 000 |
|---|---|---|---|---|---|---|---|
| `integr_gauche` | | | | | | | |
| `integr_droite` | | | | | | | |

TABLE 3.1 – Valeur de l'intégrale en fonction de $n$ pour la méthode des rectangles.

## 3.3 Méthode des trapèzes

Il existe beaucoup d'autres méthodes d'intégration numérique dont l'une, la méthode des trapèzes, est une extension directe de la méthode des rectangles. Elle consiste à remplacer les rectangles par des trapèzes dont les deux sommets s'appuient sur la courbe de la fonction $f$.

La surface du trapèze ainsi constitué est la moyenne des surfaces des rectangles obtenus avec les deux méthodes des rectangles.

**3.5** ▷ Déduisez-en à l'aide des équations 3.1 que :

$$\int_a^b f(x)dx \approx \frac{1}{2}\left[f(a) \times \frac{b-a}{n} + 2f(x_1) \times \frac{b-a}{n} + \cdots + 2f(x_{n-1}) \times \frac{b-a}{n} + f(b) \times \frac{b-a}{n}\right]$$

**3.6** ▷ Écrivez en pseudo-code l'algorithme permettant de calculer l'intégrale de la fonction $f(x) = x^2$ en utilisant la méthode des trapèzes. L'utilisateur choisira les valeurs de $a$, $b$ et $n$. Faites vérifier le pseudo-code par l'enseignant avant de poursuivre.

**3.7** ▷ Écrivez le programme correspondant à l'algorithme de la question précédente. Vous enregistrerez le programme sous le nom `trapezes.py`. La valeur de l'intégrale sera affichée avec cinq chiffres après la virgule.

**3.8** ▷ Optimisez le code de votre programme pour accélérer la vitesse de traitement, à la manière de ce que vous avez fait pour la méthode des rectangles.

**3.9** ▷ Vérifiez que la méthode des trapèzes est plus exacte que celle des rectangles pour un même nombre d'intervalles n. Pour cela, vous remplirez le tableau 3.2. Comparez les valeurs avec celles du tableau 3.1 et commentez.

| n | 1 | 2 | 4 | 50 | 100 | 1 000 | 10 000 |
|---|---|---|---|---|---|---|---|
| `integr` | | | | | | | |

TABLE 3.2 – Valeur de l'intégrale en fonction de $n$ pour la méthode des trapèzes

## Bonus

**3.10** ▷ modifiez votre programme pour que $f$ soit définie à l'aide d'une fonction que vous placerez au début du programme.

**3.11** ▷ modifiez votre programme pour que le calcul de l'intégrale soit réalisé par une fonction que vous appellerez `trapeze` et qui aura pour paramètres `a`, `b` et `n`.

Sujet de TP 4

# Entrées / sorties et fonctions personnalisées

Dans ce TP, vous apprendrez à écrire dans un fichier texte le résultat des calculs effectués par la machine. La seconde partie concerne la découverte des fonctions, notion particulièrement importante en programmation. Vous mettrez en œuvre des fonctions personnalisées

## 4.1 Lire et écrire dans des fichiers

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même, comme nous l'avons fait lors de la séance précédente pour le calcul de la moyenne (voir page ). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'information plus importante.

Il est temps que vous appreniez à séparer les données, et les programmes qui les traitent, dans des fichiers différents. Pour que cela devienne possible, nous devrons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer ensuite.

### 4.1.1 Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver (à l'aide de son titre), puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations.

Tout ce que nous venons de dire des livres s'applique aussi aux fichiers informatiques. Un fichier se compose de données enregistrées sur votre disque dur, sur une clé USB ou sur un CD-ROM. Vous y accédez grâce à son nom.

### 4.1.2 Écriture séquentielle dans un fichier

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un objet `file` qu'il faut d'abord créer. Vous pouvez ensuite lire et écrire dans ce fichier en utilisant les *méthodes* spécifiques de cet objet [1].

L'exemple ci-dessous vous montre comment ouvrir un fichier en écriture, y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Par contre, si le nom utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que vous y enregistrerez viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Faites tout cet exercice directement dans le Shell Python :

1. En programmation objet, on appelle *méthodes* les fonctions qu'il est possible d'appeler pour l'objet considéré (ici, l'objet `file`).

```
>>> f = file('monFichier.txt','a')
>>> f.write('Bonjour, fichier !')
>>> f.write("Quel beau temps, aujourd'hui !")
>>> f.close()
>>>
```

Ouvrez maintenant `monFichier.txt` dans l'environnement Wing-IDE et contrôlez son contenu. Remarquez que la fonction `write()` n'implique pas un retour à la ligne une fois le texte écrit dans le fichier. Pour revenir à la ligne, il faut ajouter le caractère `\n`, comme dans l'exemple suivant :

```
>>> f.write('Bonjour, fichier !\n')
```

**Remarques**

— La première ligne crée un objet de type `file`, lequel fait référence à un fichier véritable (sur disque ou sur un support amovible). Le nom de l'objet est `f`, tandis que le nom du fichier sur le disque est `monFichier.txt`. Ne confondez pas ces deux noms. À la suite de cet exercice, vous pouvez vérifier qu'il s'est bien créé sur votre système (dans le répertoire courant) un fichier dont le nom est `monFichier.txt` (et vous pouvez en visualiser le contenu à l'aide d'un éditeur de texte comme le bloc note).

— La création de l'objet de type `File` attend deux arguments, qui doivent être des chaînes de caractères. Le premier paramètre est le nom du fichier à ouvrir, et le second est le mode d'ouverture. « `a` » indique qu'il faut ouvrir ce fichier en mode « ajout » (append), ce qui signifie que les données à enregistrer doivent être ajoutées à la fin du fichier, à la suite de celles qui s'y trouvent éventuellement déjà. Nous aurions pu utiliser aussi le mode « `w` » (pour write), mais lorsqu'on utilise ce mode, Python crée toujours un nouveau fichier (vide), et l'écriture des données commence à partir du début de ce nouveau fichier. S'il existe déjà un fichier de même nom, celui-ci est effacé au préalable.

— La méthode `write()` réalise l'écriture proprement dite. Les données à écrire doivent être fournies en paramètre. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier à accès séquentiel). Chaque nouvel appel de `write()` continue l'écriture à la suite de ce qui est déjà enregistré.

— La méthode `close()` referme le fichier. Celui-ci est désormais disponible pour tout autre usage.

### 4.1.3 Le retour des lapins

**4.1** ▷ Copiez dans le dossier `TP4` le fichier `fibonacci.py` qui se trouve dans le dossier `TP2`. Modifiez ce programme pour qu'il calcule les 1000 premiers termes de la suite de Fibonacci et les enregistre dans un fichier texte appelé `valeurs_fibonacci.txt` (une valeur par ligne).

Attention, la méthode `write()` attend comme paramètre une chaîne de caractères. Pour écrire le contenu de la variable `chaine` dans le fichier, vous utiliserez la ligne suivante :

```
1 f.write("%d\n" % chaine)
```

il s'agit d'une *sortie formatée*, c'est-à-dire que l'on contrôle la façon d'écrire dans un fichier la valeur de `chaine`. `"%d\n"` est la chaîne de formatage. Elle indique qu'il faut écrire un entier grâce au *marqueur* `%d`, suivi d'un retour à la ligne (\n). La valeur entière à écrire est celle de la variable située après le % qui suit la chaîne de formatage.

Consultez l'annexe C page  pour connaître les autres marqueurs ainsi que la façon de contrôler finement l'écriture des données. Ouvrez ensuite le fichier `valeurs_fibonacci.txt` dans Wing-IDE et vérifiez son contenu.

## 4.2 Introduction aux fonctions

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de fonctions originales.

### 4.2.1 Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (ces sous problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

Les *fonctions* sont des structures de sous-programmes qui ont été imaginées par les concepteurs des langages afin de résoudre les difficultés évoquées ci-dessus. Nous allons décrire ici la définition de fonctions sous Python.

Nous avons déjà rencontré diverses fonctions préprogrammées, comme `sqrt()`, `round()` ou encore `len()`. Voyons à présent comment en définir nous-mêmes de nouvelles.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nom_de_la_fonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

et en pseudo-code :

```
FONCTION  nom_de_la_fonction( ( paramètres )
    |  ...
    |  bloc d'instructions
    |  ...
FIN_FONCTION
```

Algorithme 1: `fonction`

— Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis). Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom.

— Comme les instructions `if` et `while` que vous connaissez déjà, l'instruction `def` est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un deux-points, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.

— La liste de paramètres spécifie quelles informations il faudra fournir en guise de paramètre lorsque l'on voudra utiliser cette fonction (les parenthèses peuvent éventuellement rester vides si la fonction ne nécessite pas de paramètres).

— Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un appel de fonction est constitué du nom de la fonction suivi de parenthèses.

Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un paramètre pour chacun des paramètres spécifiés dans la définition de la fonction

— Une fois que la fonction a terminé son travail, elle renvoie très souvent un résultat au programme l'ayant appelé. Ce résultat (valeur numérique, texte…) doit alors être stockée dans une variable, ou utilisée directement dans une expression.

### 4.2.2 Exemple

Voici un programme qui définit la fonction `maxi(a,b)` qui renvoie au programme principal la plus grande valeur du couple $(a, b)$. Dans le programme principal (défini à la suite de la fonction), on demande à l'utilisateur de saisir deux nombres `n1` et `n2`, puis on calcule le maximum de ces deux nombres en appelant la fonction maxi définie plus haut. `n1` et `n2` sont les paramètres passés à la fonction. Celle-ci renvoie le résultat (instruction `return`) au programme principal, à l'endroit où la fonction a été appelée.

Voici l'algorithme en pseudo-code du programme :

```
# programme de détermination de la valeur maximale

FONCTIONS
 FONCTION  maxi (  a , b  )
     | SI  (a > b ) ALORS
     |    | retourner a
     | SINON
     |    | retourner b
     | FIN_SI
 FIN_FONCTION
VARIABLES
 n1,n2 DE_TYPE NOMBRE
 maximum DE_TYPE NOMBRE

DEBUT_ALGORITHME

 AFFICHER  "n1 = "
 LIRE  n1
 AFFICHER  "n2 = "
 LIRE  n2

 maximum ← maxi(n1,n2)
 AFFICHER  "le maximum est : ",maximum

FIN_ALGORITHME
```

puis le code source Python correspondant :

```
1  # coding:latin1
2  # programme de détermination de la valeur maximale
3  # Agathe Veublouz
4  # TP2, le 12/11/2013
5  # maxi.py
6  #
7  # Variables :
8  # n1 : réel, premier nombre à comparer
9  # n2 : réel, second nombre à comparer
10 # maximum, réel : contient la plus grande valeur de n1 et de n2
11
12 ###################################
13 # définition de la fonction maxi #
14 ###################################
15 def maxi(a,b):      # a et b sont des paramètres formels
16
17     # Cette fonction renvoie le maximum des
18     # deux nombres passés en paramètres
19
```

```
20      if(a>b):
21          return a
22      else:
23          return b
24
25  ##############################
26  # début du programme principal #
27  ##############################
28
29  print "Ce programme détermine le plus grand nombre parmi deux"
30  n1 = input("n1 = ")
31  n2 = input("n2 = ")
32
33  # appel de la fonction et stockage du résultat dans la variable maximum
34  # la variable dans laquelle on stocke ne doit pas porter
35  # le même nom que la fonction appelée.
36  maximum = maxi(n1,n2)
37
38  # affichage du maximum
39  print "le maximum est",maximum
```

Notez qu'on aurait pu aussi écrire directement :

```
1  print "le maximum est",maxi(n1,n2)
```

et se passer ainsi de la variable `maximum`.

**4.2** ▷ recopiez le code source Python dans le fichier `maxi.py` et testez-le. Faites constater à l'enseignant le bon fonctionnement du programme.

### 4.2.3 Fonction norme

**4.3** ▷ Écrivez un programme qui demande à l'utilisateur les coordonnées $x, y, z$ d'un vecteur $\overrightarrow{V}$, puis qui en calcule la norme définie par :

$$\|\overrightarrow{V}\| = \sqrt{x^2 + y^2 + z^2}$$

grâce à une fonction `norme` de paramètres formels $a$, $b$ et $c$, que vous écrirez. Le résultat sera alors affiché à l'écran. Vous enregistrerez le programme dans le dossier `TP4` sous le nom `norme.py`. Faites vérifier à l'enseignant le bon fonctionnement de votre programme.

## 4.3 Moyenne d'une série de valeurs

Reprenons le tableau des tensions mesurées :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 234,7 | 234,9 | 234,8 | 234,6 | 234,7 | 235,0 | 235,0 | 234,7 | 234,8 | 234,8 |
| 234,7 | 235,0 | 234,8 | 234,7 | 234,9 | 234,9 | 234,9 | 235,0 | 234,8 | 234,9 |

TABLE 4.1 – Indications mesurées ($l_v/V$)

L'objectif est d'écrire un programme qui calcule la moyenne des valeurs de ce tableau en utilisant une fonction `calculeMoyenne()`. Cette fonction prendra comme paramètre la liste des tensions, et retournera la valeur moyenne. Les tensions seront placées en début de programme dans une liste. Le programme définira la liste des tensions, et se contentera d'appeler la fonction `calculeMoyenne()` et d'afficher le résultat à l'écran.

**4.4** ▷ Écrivez le pseudo-code de la fonction permettant de déterminer la valeur moyenne des données contenues dans la liste.

**4.5** ▷ Écrivez le programme correspondant en python et testez-le.

**4.6** ▷ Modifiez le contenu de la fonction `calculeMoyenne()` en utilisant la structure de contrôle `For ... in ...`

## 4.4 Portée des variables

Vous allez apprendre dans cette partie jusqu'où est connue une variable dans un programme. Vous travaillerez dans le dossier `TP4` et vous y testerez les programmes que nous vous proposons ici.

Nous allons travailler avec une variable `x` et une procédure `affiche_x()`.

Le programme suivant :

```
1 def affiche_x():
2   x = 2
3   print x
4
5 affiche_x()
6 print x
```

donne a l'exécution :

```
2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

La valeur de `x` est correctement affichée dans la procédure (ligne 3). La variable affichée est alors la variable **locale** définie ligne 2. De retour dans le programme principal, l'affichage de la variable `x` génère une erreur (`name 'x' is not defined`) car `x` n'existe que dans la procédure `affiche_x()`

Essayons alors un autre exemple :

```
1 def affiche_x():
2         x = 2
3         print x
4
5 x = 1
6 print x
7 affiche_x()
8 print x
```

donne à l'exécution :

```
1
2
1
```

`x` est définie dans deux endroits. Elle vaut 2 dans la procédure, et 1 dans le programme principal. Bien que le même nom soit utilisé, il s'agit bien de deux variables différentes. L'une ou l'autre sera utilisée, selon l'endroit dans lequel on se trouve (dans le programme principal ou dans la procédure).

Essayons maintenant de ne plus définir x dans la procédure :

```
1 def affiche_x():
2         print x
3
4 x = 1
5 print x
6 affiche_x()
7 print x
```

Nous obtenons alors a l'exécution :

```
1
1
1
```

Surprise ! La procédure `affiche_x()` ne définit plus de variable locale `x`, et utilise alors la valeur données dans le programme principal, bien que nous n'ayons pas passé la valeur de `x` en paramètre à la procédure ! La procédure connaît la variable `x` du programme principal car cette variable a été définie *avant* l'appel à la procédure [2].

Essayons de modifier la variable :

```
1 def affiche_x():
2         x = 2
3         print x
4
5 x = 1
6 print x
7 affiche_x()
8 print x
```

donne à l'exécution :

```
1
2
1
```

Nous retombons en réalité sur l'exemple vu plus haut. Dès que l'on essaie de modifier la variable du programme principal, nous créons en fait une variable locale à la procédure. Il est donc impossible de modifier les variables du programme principal dans les procédures et fonctions, à moins de les renvoyer au programme principal (return) et de les réaffecter à la variable, comme ceci :

```
1 def affiche_x():
2         x = 2
3         print x
4         return x
5
6 x = 1
7 print x
8 x = affiche_x()
9 print x
```

ce qui donne à l'exécution :

```
1
2
2
```

Ça fonctionne, effectivement, mais ce n'est pas très pratique, voire pas pratique du tout si la fonction modifie plusieurs variables du programme principal ! (dans ce cas, il faut renvoyer une liste de variables, et les réaffecter une par une aux variables du programme principal).

Heureusement, il existe le mot clé **global**, qui indique à la procédure ou fonction quelles sont les variables du programme principal à utiliser :

```
1 def affiche_x():
2         global x # x est celui du programme principal
3         x = 2 # on modifie donc le x du programme principal
4               # sans créer de variable locale à la fonction
5         print x
6
7 x = 1
```

2. Ce ne sera pas forcément toujours le cas, il faut alors passer la variable en paramètre à la procédure.

```
8  print x
9  affiche_x()
10 print x
```

Ce qui donne bien :

```
1
2
2
```

# Deuxième partie

# Projet

# Recommandations

Les quatre séances de travaux pratiques de la partie apprentissage sont maintenant terminées. Vous avez vu les bases de la programmation, et avez suffisamment de connaissances pour commencer à développer des programmes plus conséquents. Les différentes parties suivantes correspondent au projet de la série de travaux pratiques.

L'objectif est de vous faire travailler sur un programme plus long et plus complexe, bien que s'appuyant sur les notions vues précédemment. Vous utiliserez largement les fonctions, étudiées dans la dernière séance de la partie apprentissage. Vous apprendrez également comment documenter correctement votre programme.

Le découpage est également différent, puisque bien que comportant quatre parties, celles-ci ne correspondent pas forcément au découpage en quatre séances de travaux pratiques. Vous avancerez à votre rythme, et il est probable que vous passiez plus de temps sur certaines parties du projet que sur d'autres. À vous de gérer au mieux votre temps pour tirer le meilleur parti des 13 heures de projet prévues.

Vous serez bien plus autonome également. Le sujet que nous vous fournissons ici n'est pas aussi dirigé que la partie apprentissage. Certaines informations nécessaires ne se trouvent pas dans le fascicule, et vous devrez faire quelques recherches sur internet. En particulier, nous vous encourageons vivement à consulter la documentation officielle de Python, disponible à l'adresse `https://docs.python.org/2/`. Nous ne vous demandons pas de rédiger de compte-rendu. À vous de prendre les notes nécessaires lorsque cela s'impose. Vous serez noté sur le résultat final (programme et documentation générée).

Nous vous recommandons fortement d'effectuer des sauvegardes régulières de votre travail. À la fin de chaque séance, vous créerez une archive zip contenant votre dossier de projet, nommé sur le modèle `projet_prenom_nom_date.zip`. Déposez cette archive sur claroline, dans le cours `M1204 Algorithmique et informatique`, rubrique `Travaux`. Vous serez tenu(e) responsable de toute perte de données si vous n'effectuez pas ces sauvegardes.

Bon travail !

Projet – partie 1

# Programme de base

Cette première partie se déroule en deux temps. Après avoir préparé votre environnement de travail, vous allez d'abord apprendre à lire des données texte, puis numérique, contenues dans un fichier, et à les importer dans une liste. Puis, dans un second temps, vous démarrerez le programme de votre projet proprement dit. À la fin de cette partie, vous aurez un programme comportant la grande majorité des fonctionnalités attendues. Dans les parties suivantes, vous ne ferez qu'améliorer ce programme de base, jusqu'aux dernières finitions en partie 4.

## 1.1 Préparation de l'environnement de travail

Avant toute chose, vous allez préparer les dossiers dans lesquels vous allez travailler. Placez vous dans le dossier `P:\TP_INFO_MP1`, et créez les deux dossiers suivants :

**`tests`** : ce dossier sera utilisé pour tester en python les nouvelles notions que vous allez découvrir. Un sous-dossier par notion sera créé dans `tests`

**`projet`** : il s'agit du dossier principal de votre projet. Les différents fichiers qui vont composer votre programme final se trouveront dans ce dossier.

## 1.2 Lire des données depuis un fichier

Vous allez apprendre à lire ligne par ligne dans un fichier texte. Pour cela, créez un nouveau fichier depuis Wing IDE (bouton `Nouveau` ou `Ctrl + N`). Ajoutez trois ou quatre lignes de texte (en revenant à la ligne à chaque fois avec la touche entrée), et enregistrez ce fichier dans le dossier `tests/lecture` sous le nom `blabla.txt`.

Nous avons vu dans la première partie comment ouvrir un fichier en écriture. L'ouverture en lecture est tout aussi simple. Placez-vous dans le Shell Python, et saisissez la ligne suivante :

```
fichier = file('blabla.txt','r')
```

Le `r` signifie que le fichier est ouvert en lecture (read). Pour lire la première ligne du fichier, saisissez :

```
fichier.readline()
```

**1.1** ▷ Qu'obtenez-vous ? Pourquoi la ligne lue est-elle entourée de guillemets simples ? Que signifie le \n à la fin de la ligne ?

**1.2** ▷ Exécutez cette commande plusieurs fois. Que se passe-t-il ?

Fermez le fichier, et rouvrez-le :

```
fichier.close()
fichier = file('blabla.txt','r')
```

Nous sommes prêts à relire la première ligne du fichier. Faites-le comme suit :

```
>>> texte = fichier.readline()
```

**1.3** ▷ Plus rien ne s'affiche à l'écran. Que contient la variable `texte` ?

Voici le pseudo-code qui permet de lire chaque ligne du fichier `blabla.txt` et qui l'affiche à l'écran.

```
# lecture d'un fichier texte
VARIABLES
 ligne DE_TYPE CHAINE # ligne lue
 fichier DE_TYPE objet fichier


DEBUT_ALGORITHME
 # ouverture du fichier
 fichier ← file('blabla.txt','r')

 Pour ligne dans fichier FAIRE
    AFFICHER  ligne

 Fin Pour
 # Fermeture du fichier
 fichier.fermer()
FIN_ALGORITHME
```

**1.4** ▷ Écrivez le programme correspondant à l'algorithme précédent. Vous utiliserez la structure de contrôle `for ... in ...`. Vous enregistrerez le fichier sous le nom `lecture_blabla.py` dans le dossier `tests/lecture`.

**1.5** ▷ Expliquez pourquoi chaque ligne de texte est séparée par une ligne vide.

Notez que vous pouvez supprimer le retour à la ligne (\n) à la fin de la chaîne de caractères `ligne` en remplaçant `print ligne` par `print ligne.strip()`

## 1.3 Manipuler des données dans une liste

En Python, une liste est définie en séparant les éléments par des virgules et en les enfermant par des crochets. Par exemple, nous avons défini dans la première partie la liste suivante :

```
1 tension = [ 234.7 , 234.9 , 234.8 , ...]
```

### Création et remplissage

Bien évidemment, cette méthode pour remplir une liste n'est pas utilisable si les données se trouvent dans un fichier (il est hors de question de recopier les données à la main au début du programme !). Au lieu de cela, il faut d'abord créer une liste vide, puis ajouter les éléments à la liste en utilisant la méthode `append()`. Testez les lignes suivantes dans le Shell Python :

```
>>> tension = []  # crée la liste vide
>>> tension.append(234.7) # on ajoute des éléments à la liste
>>> tension.append(234.9)
>>> tension.append(234.8)
>>> print tension # on affiche le contenu de la liste
[234.7, 234.9, 234.8]
```

### Longueur d'une liste

La fonction `len()` renvoie le nombre d'éléments présents dans la liste :

```
>>> len(tension)
3
```

### Suppression d'un élément

Une autre fonction intégrée permet de supprimer d'une liste un élément quelconque à partir de son index. Il s'agit de `del()` :

```
>>> del(tension[1])
>>> print tension
[234.7, 234.8]
```

### Ajout d'un élément

Il est possible ensuite d'ajouter un élément à la liste à l'aide de la méthode `insert(p,e)` ou `p` est la position dans la liste, et `e` est l'élément à ajouter. Par exemple :

```
tension.insert(1,234.9)
>>> tension
[234.7, 234.9, 234.8]
```

### Parcourir une liste avec `for...in...` :

Pour terminer, notez que vous pouvez facilement accéder de manière consécutive à tous les éléments d'une liste en utilisant l'instruction `for ... in ...` vue précédemment :

```
>>> for t in tension:
...     print t
...
234.7
234.9
234.8
```

`t` prend alors consécutivement toutes les valeurs de la liste.

### `sort()` et `reverse()`

**1.6** ▷ Expérimentez ces deux méthodes dans le Shell Python et expliquez leurs rôles.

La documentation des méthodes disponibles pour les listes se trouve à l'adresse :
`http://docs.python.org/2/tutorial/datastructures.html`

## 1.4 Lecture d'un fichier de points

L'enseignant vous a fourni un fichier nommé `signal.dat`. Copiez-le dans le dossier `tests/lecture` ainsi que dans le dossier `projet`. Il s'agit du signal électrique que vous souhaitez analyser. Les tensions ont été enregistrées par un oscilloscope numérique, puis exportées sur clé usb.

**1.7** ▷ La fréquence d'échantillonnage était de $f_e = 1$ kHz. Quel est l'intervalle de temps $\Delta t$ séparant deux échantillons ?

**1.8** ▷ Commencez par ouvrir `signal.dat` dans Wing IDE et décrivez son contenu. Combien comporte-t-il de lignes ? Quelle a été la durée de l'enregistrement ?

**1.9** ▷ Écrivez en pseudo-code l'algorithme capable d'ouvrir ce fichier de données, et de placer son contenu dans une liste nommée `v` pour y effectuer un traitement ultérieur. Faites vérifier l'algorithme par l'enseignant avant de poursuivre.

**1.10** ▷ Écrivez le programme correspondant à l'algorithme précédent. Vous enregistrerez votre programme dans le dossier `tests/lecture` sous le nom `lecture_donnees.py`.

**1.11** ▷ Quel est le type de données contenu dans la liste ? Expliquez pourquoi cela pose un problème pour traiter les valeurs des tensions.

**1.12** ▷ Pour convertir une chaîne de caractères en un nombre réel, il faut utiliser la fonction `float()`. Testez l'exemple suivant dans le Shell Python :

```
>>> print float('1.2\n')
1.2
```

La chaîne de caractères `'1.2\n'` a bien été convertie en réel `1.2`. Notez qu'avec cette fonction, il n'est plus nécessaire d'utiliser la méthode `strip()` pour supprimer le retour à la ligne `\n`. Modifiez votre programme pour qu'il stocke dans la liste `v` non pas des chaînes de caractères, mais des réels.

**1.13** ▷ Il est très facile de tracer les valeurs des tensions en utilisant la bibliothèque `pylab`. Pour cela, ajoutez au début de votre programme :

```
1 from pylab import plot,show
```

et à la fin de votre programme :

```
1 plot(v)
2 show()
```

Maintenant que les données sont représentées à l'écran, estimez graphiquement l'amplitude crête $\hat{V}$ ainsi que la période $T$ en secondes. Ouvrez ensuite le fichier `signal.dat` dans Wing IDE et déterminez de manière exacte l'amplitude crête et la période.

## 1.5 Analyse des données

Maintenant que votre programme sait lire les données stockées dans un fichier texte et les importer dans une liste, vous allez analyser ces données en déterminant quelques caractéristiques du signal.

### 1.5.1 Calcul de la valeur moyenne

La valeur moyenne d'un signal $v(t)$ de période $T$ est donnée par :

$$\overline{V} = \frac{1}{T}\int_0^T v(t)dt$$

Notez que la valeur moyenne de ce signal périodique doit être calculée en intégrant exactement sur une période $T$. La valeur moyenne peut également être approximée par la somme discrète :

$$\overline{V} = \frac{1}{T}\sum_{i=0}^{n-1} v_i \Delta t = \frac{\Delta t}{T}\sum_{i=0}^{n-1} v_i$$

où $n$ représente le nombre d'échantillons par période du signal et $v_i$ est la i$^{e}$ valeur.

**1.14** ▷ Écrivez en pseudo-code l'algorithme du programme calculant la moyenne du signal **sur une période**. Faites vérifier par l'enseignant avant de poursuivre.

**1.15** ▷ Écrivez le programme en repartant de `lecture_donnees.py`. Enregistrez-le sous le nom `analyse.py` dans le dossier `projet`.

**1.16** ▷ Vérifiez que la valeur moyenne de la tension est proche de :

$$\overline{V} = \frac{2}{\pi}\hat{V}$$

**1.17** ▷ Calculez l'écart relatif entre la valeur moyenne que vous avez calculée et la valeur exacte donnée par la relation ci-dessus. Pourquoi votre programme ne donne-t-il pas la valeur moyenne correcte ?

Remarquez au passage que calculer la valeur moyenne de la tension revient à calculer une valeur approchée de l'intégrale de $v(t)$ en utilisant la méthode des rectangles vue dans la séance précédente. Les bornes d'intégration sont ici $a = 0$ et $b = T$, et la largeur d'un rectangle vaut $h = \Delta T$.

### 1.5.2 Calcul de l'écart-type

Continuez de travailler dans le fichier `analyse.py`.

En vous aidant de ce qui a été vu dans la partie apprentissage de la série de TP, complétez votre programme par le calcul de l'écart-type du signal (calculé sur une période). Vous réutiliserez la valeur moyenne calculée précédemment.

### 1.5.3 Calcul de la composante alternative du signal

**1.18** ▷ Donnez la relation entre $v(t)$, $\overline{V}$ et $v_{alt}(t)$, la composante alternative du signal.

**1.19** ▷ Écrivez en pseudo-code l'algorithme du programme qui calcule la composante alternative `Valt` de la tension `v`. Vous réutiliserez la valeur moyenne calculée précédemment. Vous utiliserez également la méthode `ajouter` (`append()`) ainsi que `Pour...dans...` (`For...in...`). Faites vérifier par l'enseignant avant de poursuivre.

**1.20** ▷ Complétez votre programme en y ajoutant le code python correspondant à l'algorithme que vous avez écrit.

**1.21** ▷ Faites vérifier à l'enseignant pour valider la partie 1 du projet.

Projet – partie 2

# Modules et fonctions personnalisées

Vous avez développé dans la partie précédente un programme (`analyse.py`) qui contient les fonctionnalités suivantes :

— lecture d'un fichier de point et importation dans une liste,
— calcul de la moyenne sur une période,
— calcul de l'écart-type expérimental,
— calcul de la composante alternative.

Avant toute chose, le sujet de TP n°4 de la partie apprentissage (consacré en grande partie à l'étude des fonctions) contient des notions essentielles pour réaliser cette partie du projet. Si vous n'avez pas pu terminer ce sujet, il est vivement recommandé de le terminer avant de démarrer cette partie. Si vous l'aviez terminé, prenez tout de même quelques minutes pour relire la partie « Introduction aux fonctions » page .

## 2.1 Introduction aux modules

Un module est un morceau de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à importer le module et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module math qui contient, comme son nom l'indique, des fonctions mathématiques. et nous verrons les différentes méthodes d'importation des modules dans vos programmes.

### 2.1.1 La méthode import

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module math ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
1 >>> import math
2 >>>
```

La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici `math`.

Après l'exécution de cette instruction, rien ne se passe… en apparence. En réalité, Python vient d'importer le module math. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math.sqrt(16)
4
>>>
```

Comme vous le voyez, la fonction `sqrt` du module `math` renvoie la racine carrée du nombre passé en paramètre.

Pour savoir quelles fonctions existent et ce qu'elles font, il faut utiliser la fonction `help`, qui prend en paramètre la fonction ou le module sur lequel vous demandez de l'aide. L'aide est fournie en anglais mais c'est de l'anglais technique, c'est-à-dire une forme de l'anglais que vous devrez maîtriser pour programmer, si ce n'est pas déjà le cas. Une grande majorité de la documentation est en anglais, bien que vous puissiez maintenant en trouver une bonne part en français.

```
>>> help("math")
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians) of x.

    asin(...)
-- Suite  --
```

Si vous parlez un minimum l'anglais, vous avez accès à une description exhaustive des fonctions du module `math`. Vous voyez en haut de la page le nom du module, le fichier qui l'héberge, puis la description du module. Ensuite se trouve une liste des fonctions, chacune étant accompagnée d'une courte description.

Tapez `Q` pour revenir à la fenêtre d'interpréteur, `Espace` pour avancer d'une page, `Entrée` pour avancer d'une ligne. Vous pouvez également passer un nom de fonction en paramètre de la fonction `help`.

```
>>> help("math.sqrt")
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

```
>>>
```

### 2.1.2 La méthode from ... import ...

Il existe une autre méthode d'importation, que vous avez déjà vue dans la partie apprentissage, et qui ne fonctionne pas tout à fait de la même façon. Reprenons notre exemple du module math. Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
1 >>> from math import fabs
2 >>> fabs(-5)
3 5
4 >>> fabs(2)
5 2
6 >>>
```

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « * » à la place du nom de la fonction à importer.

```
1 >>> from math import *
2 >>> sqrt(4)
3 2
4 >>> fabs(5)
5 5
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module math et les a importées directement. Il n'est plus nécessaire d'écrire `math.sqrt()`, on peut utiliser directement `sqrt()`

## 2.2 Comment créer ses propres modules

### 2.2.1 Création et importation du module

Commencez par vous créer un dossier `tests/module`. Nous allons créer deux fichiers `.py` dans ce dossier :

— un fichier `multipli.py`, qui contiendra la fonction `table`;
— un fichier `principal.py`, qui contiendra le programme principal qui fera appel à notre module.

N'oubliez pas de spécifier la ligne précisant l'encodage en tête de vos deux fichiers. Maintenant, voyons le code du fichier `multipli.py`.

```
1 """module multipli contenant la fonction table"""
2
3 def table(nb):
4     """Fonction affichant la table de multiplication par nb de
5     1 * nb jusqu'à 10 * nb"""
6     i = 0
7     while i < 10:
8         print i + 1, "*", nb, "=", (i + 1) * nb
9         i += 1
```

On se contente de définir une seule fonction, `table`, qui affiche la table de multiplication choisie. Le texte entre trois guillemets doubles (`"""comme ceci"""`) correspond à la documentation du module, ou de chaque fonction. On parle alors de docstrings (chaînes de documentation). Testez les lignes suivantes dans un shell Python :

```
>>> from multipli import *
>>> help(table)
>>> help('multipli')
```

Il vous faudra systématiquement commenter vos modules et fonctions.

Voici le code du fichier `principal.py`, n'oubliez pas la ligne précisant votre encodage, en tête du fichier.

```
import os
from multipli import *

# test de la fonction table
table(3)
os.system("pause")
```

En le lançant directement, voilà ce qu'on obtient :

```
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
10 * 3 = 30
Appuyez sur une touche pour continuer...
```

Nous avons vu comment créer un module, il suffit de le mettre dans un fichier. On peut alors l'importer depuis un autre fichier contenu dans le même répertoire en précisant le nom du fichier (sans l'extension `.py`).

## 2.3 Création du module de fonctions

Il est temps maintenant de créer votre propre module de fonctions.

**2.1** ▷ Créez un fichier `fonctions.py` dans le dossier `projet`. C'est dans ce fichier que nous allons placer les différentes fonctions utilisées dans le projet.

### 2.3.1 Écriture des fonctions personnalisées

**2.2** ▷ dans le fichier `fonctions.py`, créez la fonction `lecture_donnees( fichier )`, qui prend comme paramètre une chaîne de caractère contenant le nom du fichier contenant les données que l'on veut lire, et qui renvoie une liste contenant les valeurs numériques qui étaient contenues dans le fichier. Créez les variables locales à la fonction nécessaires à son bon fonctionnement. N'oubliez pas d'indiquer en commentaires au début de chaque fonction les variables utilisées, leur type et leur rôle dans la fonction. Faites ceci pour les variables locales et celles passées en paramètre. Faîtes vérifier la fonction par l'enseignant.

**2.3** ▷ de la même manière, créez la fonction `valeur_moyenne( v , deltat , T)`, qui prend comme paramètre la liste des tensions, la période d'échantillonnage `deltat` et la période du signal `T`. Cette fonction renvoie la valeur moyenne de `v`. De la même manière (et pour les questions suivantes également), créez les variables locales nécessaires et indiquez le rôle de chacune d'entre-elles.

**2.4** ▷ créez la fonction `ecart_type( v , deltat , T)`, qui prend comme paramètre la liste des tensions, la période d'échantillonnage `deltat` et la période du signal `T`. Cette fonction renvoie l'écart-type des valeurs de `v`.

**2.5** ▷ enfin, créez la fonction `composante_alt( v , deltat , T)`, qui prend comme paramètre la liste des tensions, la période d'échantillonnage `deltat` et la période du signal `T`, et qui renvoie la composante alternative du signal contenu dans la liste `v`.

### 2.3.2 Appel des fonctions

**2.6** ▷ Modifiez le programme `analyse.py` pour qu'il utilise les fonctions que vous avez placées dans le module `fonctions`. Vérifiez le bon fonctionnement de votre programme.

**2.7** ▷ Ajoutez la documentation du module en indiquant son rôle, ainsi que la documentation de chaque fonction (indiquez clairement quel est son rôle et ce qu'elle renvoie).

**2.8** ▷ Faites vérifier à l'enseignant pour valider la partie 2 du projet.

Projet – partie 3

# Documentation du projet

La qualité d'un logiciel, outre ses fonctionnalités et son efficience, dépend également de la façon dont son code est écrit et documenté. Ceci est particulièrement vrai lorsqu'il va s'agir de maintenir un logiciel (correction de bogues), ou le faire évoluer (ajout de nouvelles fonctionnalités). On distingue deux types de documentations :

— la documentation orientée utilisateur. Il s'agit ici de décrire les fonctionnalités du logiciel du point de vue de l'utilisateur. Le fonctionnement interne du logiciel est complètement passé sous silence. Bien que cette documentation soit très importante, ce n'est pas sur elle que nous allons nous concentrer en priorité.

— la documentation orientée développeur. On cherche ici à expliquer le fonctionnement interne du logiciel. Il ne s'agit pas ici de récupérer les commentaires qui se trouvent dans le code, mais plutôt de les compléter.

Pour documenter votre projet, nous n'utiliserons pas les docstrings comme vu dans le chapitre précédent (page ), mais doxygen, un outil de documentation bien plus puissant et polyvalent. Ce n'est pas le seul logiciel de génération de documentation. Une liste relativement complète peut-être trouvée ici :

`http://en.wikipedia.org/wiki/Comparison_of_documentation_generators`

## 3.1 Présentation de doxygen

Doxygen est un système de documentation pour C, C++, Java, Python, Php et autres langages. Il permet de générer la documentation de vos développements :

— à partir des commentaires insérés dans le code source

— à défaut de commentaires, à partir de la structure du code lui même. La documentation générée sera dans ce cas minimale.

La documentation peut être produite dans des formats variés tels que des pages web pour une lecture en ligne (HTML), ou encore des documents PDF pour une documentation imprimable (générés avec LaTeX).

Doxygen est un logiciel libre (son code source est disponible), multiplate-formes, et gratuit.

## 3.2 Initiation à doxygen

### 3.2.1 Préparation de l'environnement

Dans cette partie, vous allez commencer à documenter un programme, en partant d'un exemple simple.

Créez le dossier `tests/doxygen`, et copiez-y les deux fichiers `doxygen/fonctions.py` et `doxygen/resolution.py`.

**`resolution.py` :** ce programme calcule les racines d'un polynôme du second degré $ax^2 + bx + c = 0$. Les valeurs de $a$, $b$ et $c$ sont demandées à l'utilisateur. Les racines sont ensuite calculées grâce à la fonction `polynome` qui se trouve dans le module `fonctions.py`

**`fonctions.py` :** ce module contient les fonctions suivantes :

- `polynome(a,b,c)`, qui calcule les racines réelles du polynome et les renvoie dans une liste. S'il n'y a pas de solution réelle, une liste vide est renvoyée. Les trois paramètres de la fonction sont les coefficients du polynôme.
- `discriminant(a,b,c)` qui calcule et renvoie le discriminant du polynôme à partir des trois coefficients passés en paramètres.

Prenez le temps d'analyser le code contenu dans ces deux fichiers, et de bien comprendre son fonctionnement. N'hésitez pas à appeler l'enseignant pour obtenir des explications si certaines parties du programme ne sont pas comprises.

Ces deux programmes ne contiennent volontairement aucun commentaire ni documentation. Vous allez les ajouter au fur et à mesure, et générer la documentation correspondante au format html.

### 3.2.2 Configuration de doxygen

Vous allez maintenant exécuter l'assistant de doxygen (`Doxywizard`), et le configurer pour le faire travailler sur ces fichiers.

Lancez l'assistant à l'aide du raccourci `Démarrer / Programmes / doxygen / Doxywizard`. La fenêtre de l'assistant de doxygen s'ouvre alors.

- Spécifiez le dossier de travail de doxygen. Pour cela, cliquez sur `Select...` et choisissez le dossier `tests/doxygen` dans votre dossier de travail. C'est dans ce dossier que les sources python se trouvent, et que la documentation générée sera placée. Notez que nous aurions pu choisir un autre dossier.
- Pour l'étape suivante, vous trouverez quatre éléments (topics) à configurer. Vous êtes normalement déjà sur l'élément `Projet`. Dans la partie droite du panneau, configurez les éléments suivants :

  **Project name :** Polynômes

  **Project synopsis :** calcul des racines réelles

  **Project version or id :** 1.0
- Puisque le dossier de travail de doxygen correspond également au dossier contenant les sources, il est inutile de remplir le champ `Source code directory`. `Scan recursively` demande à doxygen d'analyser les fichiers sources qui se trouveraient dans les sous-dossiers du projet. Comme ici nous n'avons pas de sous-dossier, inutile de cocher cette case.
- De même, puisque nous voulons placer la documentation générée dans ce même dossier, il est inutile de remplir le champ `Destination directory`
- Cliquez ensuite sur `Next` pour passer à l'onglet suivant. Vous pouvez également cliquer directement sur le topic `Mode`. Dans le panneau de droite, choisissez simplement `Optimize for Java or C# output` puisque c'est au langage Java que Python se rapproche le plus au niveau de la syntaxe. Cliquez sur `Next`
- Vous êtes maintenant dans le topic `Output`. Vous allez choisir dans quel format vous souhaitez générer la documentation. Vérifiez que `HTML` est bien coché pour générer les pages web. Décochez `LaTeX`. Cliquez sur `Next`.
- Pour le topic `Diagrams`, aucune configuration à faire. Nous n'utiliserons pas les diagrammes de toutes façons.

Vous êtes maintenant prêt(e) à lancer doxygen. Cliquez sur l'onglet `Run`, puis sur le bouton `Run doxygen`. Tous les fichiers de documentation sont alors générés. Vérifiez qu'un dossier `html` a bien été créé dans `tests/doxygen`, et regardez son contenu. Un grand nombre de fichiers ont été générés automatiquement.

Pour voir la documentation, vous pouvez :

— cliquez sur `Show HTML output` dans l'onglet `Run` de l'assistant de doxygen,
— ou cliquez directement sur le fichier `index.html` du dossier `html` contenant la documentation.

Vous voyez alors la documentation en ligne du projet dans un navigateur web. Elle ne contient quasiment rien. Normal, vous n'avez pas encore documenté les programmes. Vous remarquez cependant que quelques termes sont en anglais (`Main Page`, `Search`...).

Revenez dans l'assistant de doxygen, puis cliquez sur l'onglet `Expert`. Configurez la langue de la documentation dans la liste déroulante `OUTPUT_LANGUAGE`. Retournez ensuite sur l'onglet Run, et cliquez à nouveau sur `Run doxygen`. Dans le navigateur web, appuyez sur la touche `F5` pour mettre à jour l'affichage. La page doit maintenant être en français.

### 3.2.3 Documentation du module

Il est maintenant temps d'ajouter de la documentation aux programmes.

Doxygen fait la distinction entre les commentaires « classiques » (commençant par le caractère #) que vous avez utilisé jusque là dans vos programme, et les commentaires qui serviront à génerer la documentation. Pour cela, il faut utiliser la syntaxe suivante :

```
##
# ces lignes seront
# considérées par doxygen
# comme faisant partie de la documentation
```

Nous allons commencer par indiquer à doxygen que le fichier `fonctions.py` est un module de fonctions servant à manipuler les polynômes du second degré. En haut de ce fichier, ajoutez le texte suivant :

```
##
# @package fonctions
# @author nom prénom
# @version 1.0
# @date 28 octobre 2013
# @brief module de fonctions de manipulation de polynômes du second degré
# @details ce module contient les fonctions permettant de calculer le
# discriminant d'un polynôme, et d'en calculer les racines réelles
```

doxygen utilise des mots clés (tags) pour donner du sens au texte ajouté dans la documentation. Par exemple, le mot clé `@package` indique que le fichier est un module de fonctions, et qu'il s'appelle `fonctions`. Attention, après le mot clé `@package`, il faut placer le nom du fichier dans lequel est enregistré le module, sans l'extension `.py`. Nous ne détaillons pas ici la signification des autres mots clés, ils ne présentent aucune difficulté.

Relancez la génération de la documentation, et observez les changements dans le navigateur web. Vous voyez maintenant apparaître une nouvelle rubrique `Paquetages`, qui correspond aux modules Python documentés. Regardez la documentation du paquetage `fonctions`. Vous y retrouverez les documentations brève (`@brief`) et détaillée (`details`) que vous avez saisies. Notez également que les fonctions et variables du module ont été automatiquement récupérées. Enfin, l'auteur du module, la version et la date apparaissent bien dans la description détaillée du module.

Il y a cependant un petit problème au niveau des accents. Cela provient d'un problème d'encodage des fichiers. Par défaut, doxygen travaille avec l'encodage utf-8, alors que vous fichiers sont encodés en latin1 (iso 8859-1). Inutile de modifier l'encodage de vos fichiers sources. Nous allons simplement indiquer à doxygen quel est l'encodage de vos fichiers.

Pour cela, retournez dans l'assistant, onglet `Expert`. Choisissez alors le topic `Input`, puis dans le champ `INPUT_ENCODING`, remplacez `UTF-8` par `LATIN1`. Regénérez la documentation et vérifiez que les accents sont correctement affichés maintenant.

Afin d'éviter de ressaisir toute la configuration de doxygen la prochaine fois, nous allons l'enregistrer dans un fichier. Cliquez sur `File / Save`, puis enregistrez le fichier de configuration sous le nom `polynome.doxy` dans votre dossier `tests/doxygen`. Vous pourrez recharger la configuration à tout moment grâce à l'entrée de menu `File / Open`.

### 3.2.4 Documentation des fonctions

Les fonctions ont bien été découvertes par doxygen, mais n'ont pas été documentées. Il est temps de remédier à cela maintenant. Ajoutez le code suivant **juste avant** la définition de la fonction `discriminant` :

```
##
# @brief calcule le discriminant
# @param a coefficient du polynôme
# @param b coefficient du polynôme
# @param c coefficient du polynôme
# @return le discriminant déterminé par delta = b*b-4*a*c
```

**`@brief:`** comme tout à l'heure, il s'agit d'une description brève de ce que fait la fonction,

**`@param`** documente un paramètre de la fonction. Ce mot clé est suivi du nom du paramètre, puis de sa signification,

**`@return`** indique ce que renvoie la fonction.

Mettez à jour la documentation, et observez les changements. De la même, manière, documentez l'autre fonction du module :

```
##
# @brief calcule les racines réelles du polynôme
# @details Cette fonction calcule les racines réelles du polynôme dont
# les coefficients sont passés en paramètres. Les racines sont renvoyées dans
# une liste. Cette liste contient :
# - deux éléments si le discriminant est positif,
# - un seul élément s'il est nul,
# - et aucun s'il est négatif.
# @param a coefficient du polynôme
# @param b coefficient du polynôme
# @param c coefficient du polynôme
# @return une liste contenant les racines réelles
```

Quelques remarques :

— la documentation peut s'étaler sur plusieurs lignes, comme ici dans la description détaillée de la fonction,

— les tirets permettent de faire des listes.

Attention, la documentation des fonctions indique ce que fait la fonction dans les grandes lignes, mais ne dispense pas d'ajouter des commentaires dans le code de la fonction. Ces commentaires ne seront pas apparents dans la documentation, mais resteront consultables dans le code par le programmeur. Ajoutez les commentaires à la fonction `polynome` :

```
1 def polynome(a,b,c):
2
3     # création d'une liste vide pour accueillir les solutions
4     solutions = []
5
6     # calcul du discriminant
7     delta = discriminant(a,b,c)
8
9     # calcul et stockage des solutions dans la liste
```

```
10    # en fonction de la valeur du discriminant
11    if(delta > 0):
12       x1 = (-b - sqrt(delta))/(2*a)
13       x2 = (-b + sqrt(delta))/(2*a)
14       solutions.append(x1)
15       solutions.append(x2)
16    elif(delta == 0):
17       x1 = -b / (2*a)
18       solutions.append(x1)
19
20    # on renvoie la liste contenant les solutions
21    return solutions
```

Regénérez la documentation, et vérifiez que ces commentaires n'apparaissent nulle part.

### 3.2.5 Documentation du programme principal

Le programme principal (`resolution.py`) n'est pas un module, et ne contient pas de fonctions. Nous allons donc le documenter différemment, en y plaçant la documentation plus orientée « utilisateur ». Pour cela, ajouter les lignes suivantes *quelque part* dans le fichier (leur emplacement n'a pas vraiment d'importance).

```
##
# @mainpage Présentation du projet
#
# Ce programme calcule les racines réelles d'un polynôme du second degré
# en résolvant l'équation ax2 + bx + c = 0. Les coefficients a, b et c
# sont demandés à l'utilisateur, puis on affiche une liste des solutions.
# @author Agathe Veublouze
# @version 1.0
# @date 28 octobre 2013
```

Régénérez la documentation et observez les changements.

### 3.2.6 Ajout de quelques améliorations

Voilà, c'est presque terminé. Nous allons ajouter quelques éléments intéressants. Dans la documentation du programme principal, ajoutez les lignes suivantes :

```
# @todo ajouter la prise en charge des solutions complexes si le
# discriminant est négatif
# @bug si les coefficiens a, b et c ne sont pas des nombres, le programme
# ne fonctionne pas
```

`@todo` permet de garder une trace de ce qu'il reste à faire dans le développement du programme. Nous l'avons placé ici dans la documentation principale, mais il est également possible de le placer dans la documentation d'un module ou d'une fonction.

`@bug` permet de lister les bogues connus afin de signaler à l'utilisateur à quoi il doit s'attendre, mais aussi pour garder une trace des bogues qu'il faudra corriger. De même, il est possible de le placer dans la documentation d'un module ou d'une fonction.

Il existe un grand nombre de mots clés. Nous n'en avons vu ici qu'une toute petite partie. La liste de tous les mots clés existant est disponible dans la documentation officielle de doxygen à l'adresse suivante :

`http://www.stack.nl/~dimitri/doxygen/manual/commands.html`

Pour terminer cette initiation à doxygen, nous allons ajouter les sources des programmes dans la documentation. Retournez dans l'assistant de doxygen, dans l'onglet `Expert`, topic `Source Browser`,

et cochez `SOURCE_BROWSER`. Une rubrique `Fichiers` apparaîtra dans la documentation, et le code source des deux fichiers python y sera directement consultable. Cochez également `INLINE_SOURCE`. Le code de chaque fonction apparaîtra alors sous sa documentation. Certains éléments du code seront alors même cliquables pour naviguer plus facilement dans la documentation.

Regénérez la documentation et observez les changements. N'oubliez pas d'enregistrer le fichier de configuration de doxygen.

## 3.3 Documentation de votre projet

Replacez-vous dans le dossier `projet`. Maintenant que vous savez utiliser doxygen, vous pouvez entièrement documenter votre projet. Vous ajouterez la documentation pour :

— le programme principal, en y expliquant les principes et le fonctionnement du programme
— le module `fonctions`
— toutes les fonctions de ce module

Vous ajouterez un logo à votre projet (topic `Project`) et choisirez une couleur dominante pour votre documentation (topic `Output`, bouton `Change color...`)

Faites vérifier à l'enseignant pour valider la partie 2 du projet.

Projet – partie 4

# Finalisation du projet

Dernière ligne droite. Votre projet est presque terminé. Dans un premier temps, vous allez le compléter avec quelques fonctions. Ensuite, vous allez le rendre un peu plus interactif avec l'utilisateur, en développant un menu et un système de procédures en fonction de l'élément de menu choisi. Enfin, vous ajouterez un peu de couleur, et finaliserez la documentation du projet.

## 4.1 Ajout de fonctions complémentaires

### 4.1.1 Enregistrement de `Valt` dans un fichier

**4.1** ▷ Écrivez la fonction `enregistre(v,fichier)` qui enregistre les valeurs de la liste `v` dans un fichier dont le nom est contenu dans la variable `fichier`. Les données seront enregistrées en colonne avec trois chiffres après la virgule.

**4.2** ▷ documentez de façon exhaustive la fonction `enregistre` : brève description, description détaillée, paramètres, bogues, alertes…

**4.3** ▷ Depuis le programme principal, enregistrez la composante alternative du signal dans un fichier nommé `alt.dat` en utilisant la fonction définie dans la question précédente.

### 4.1.2 Ajout d'un offset à une liste de valeurs

**4.4** ▷ Écrivez la fonction `ajoute_offset(v,value)`, qui prend comme paramètre la liste `v`, ajoute l'offset `value` à chacun de ses éléments, et renvoie le résultat dans une liste `Voffset`.

**4.5** ▷ documentez de façon exhaustive la fonction `ajoute_offset` : brève description, paramètres, valeur retournée…

## 4.2 Création d'un menu

L'objectif est de créer un menu pour que l'utilisateur puisse choisir ce qu'il souhaite faire avec votre programme.

Dans le programme `analyse.py`, ajoutez une procédure `menu()` qui affiche le menu suivant au lancement du programme :

```
1) charger des données
2) tracer les données
3) calculer la valeur moyenne
4) calculer l'écart-type expérimental
5) ajouter un offset
6) exporter les données
7) exporter la composante alternative
```

```
8) quitter

votre choix :
```

et qui, en fonction de la réponse de l'utilisateur, va appeler les procédures suivantes :
— `menu_charger()`
— `menu_tracer()`
— `menu_valeur_moyenne()`
— `menu_ecart_type()`
— `menu_offset()`
— `menu_exporter()`
— et `menu_exporter_alt()`

Ces procédures feront appel aux fonctions que vous avec écrites dans le module `fonctions`.

Bien entendu, il faudra ensuite afficher de nouveau le menu, jusqu'à ce que l'utilisateur choisisse de quitter le programme. Pour plus de clarté, effacez l'écran avant d'afficher le menu (recherchez sur internet comment faire ceci).

Nous vous recommandons également vivement de créer les variables suivantes dans votre programme principal :

| | |
|---|---|
| `deltat` | période d'échantillonnage, en s |
| `T` | période du signal en s |
| `fichier` | fichier dans lequel les données ont été lues |
| `fichierExportation` | fichier dans lequel les données sont enregistrées |
| `fichierExportationAlt` | fichier dans lequel la composante alternative des données sont enregistrées |
| `v` | liste des tensions lues depuis le fichier en V |
| `Vmoy` | valeur moyenne des tensions en V |
| `ecartType` | écart-type expérimental des tensions en V |
| `offset` | dernier offset appliqué en V |

Vos procédures `menu_` vont manipuler les variables du programme principal. Plutôt que de passer toutes les variables en paramètres aux procédures (qui deviendraient donc des fonctions !), et de récupérer ces variables dans une liste (return), vos procédures vont travailler directement avec les variables du programme principal. Relisez attentivement la partie sur la portée des variables (section 4.4 page ) pour comprendre comment écrire vos procédures `menu_`.

## 4.3 Menu amélioré

Modifiez le menu pour qu'en dessous de chaque entrée de menu s'affiche l'information correspondante une fois cette entrée de menu appelée, comme ceci :

```
1) charger des données
   fichier :  signal.dat
2) tracer les données
3) calculer la valeur moyenne
   moyenne :  74.12 V
4) calculer l'écart-type expérimental
   écart-type :  30.8 V
5) ajouter un offset
   offset :  10.0 V
6) exporter les données
   enregistré dans :  data.txt
7) exporter la composante alternative
```

```
    enregistré dans :  data_alt.txt
 8) quitter

votre choix :
```

Évidemment, tant qu'une entrée de menu n'a pas été appelée, on doit plutôt afficher une ligne vide en-dessous.

## 4.4 Ajout de couleur

Dans ce menu, on souhaite afficher les noms de fichiers en bleu clair (cyan) et les valeurs numériques en vert. Nous vous proposons d'utiliser le module `colorama`. Le fichier `colorama-0.3.2.zip` se trouve sur claroline. Copiez-le et décompressez-le dans un dossier temporaire, puis copiez le dossier `colorama` dans votre dossier projet. Aidez-vous de la documentation du module pour modifiez votre programme et utiliser la couleur.

Aidez-vous de la documentation de `colorama`, disponible à cette adresse :
`https://pypi.python.org/pypi/colorama`

## 4.5 Documentation

N'oubliez pas de documenter toutes les procédures, fonctions et fonctionnalités que vous avez ajoutées dans cette quatrième partie. De nouvelles procédures ayant été ajoutées dans `analyse.py`, documentez ce fichier comme un module, au même titre que `fonctions.py`.

Indiquez également les choses restant à faire pour améliorer le programme (`@todo`) ainsi que les bogues connus mais non encore corrigés (`@bug`).

## Bonus

— S'il vous reste du temps, n'hésitez pas à améliorer votre programme par l'ajout de nouvelles fonctionnalités (vous avez carte blanche), mais à chaque fois, documentez-les.

Annexe A

# Algorithmes et Python

Le tableau ci-dessous rassemble les différents types d'instructions que vous rencontrez lors de la série de travaux pratiques. À gauche se trouve l'algorithme en pseudo-code, et à droite le code correspondant en Python.

### Commentaires

```
# commentaire
```

```
# commentaire
```

### Affectation

```
x ← 1
```

```
x = 1
```

### Déclaration d'une variable

```
a DE_TYPE NOMBRE # borne inférieure
```

```
# a : NOMBRE, borne inférieure
```

### Écrire du texte à l'écran

```
AFFICHER  "Bonjour !"
```

```
print "Bonjour !"
```

### Lire une valeur

```
AFFICHER  "Saisissez v : "
LIRE  v
```

```
v = input("Saisissez v : ")
```

### Boucle Tant que (while)

```
TANT_QUE  (condition) FAIRE
    | instruction_1
    | instruction_2
FIN_TANT_QUE
instruction_3
```

```
while condition:
    instruction_1
    instruction_2
instruction_3
```

### Fonctions

```
FONCTION  nomde_la_fonction (  paramètres
)
    | ...
    | bloc d'instructions
    | ...
    | retourner  r
FIN_FONCTION
```

```
def nomde_la_fonction( paramètres ):
    ...
    bloc d'instructions
    ...
    return r
```

### Si...sinon (if...else)

```
SI  (condition) ALORS
   | instruction_1
   | instruction_2
SINON
   | instruction_3
FIN_SI
instruction_4
```

```
if condition:
   instruction_1
   instruction_2
else:
   instruction_3
instruction_4
```

### Pour...dans (for...in)

```
Pour V dans v FAIRE
   | Valt.ajouter(V - vm(v,deltaT,T))
Fin Pour
```

```
for V in v:
   Valt.append(V - vm(v,deltaT,T))
```

### Manipuler des fichiers

```
fichier ← file(nom,'r')
Pour ligne dans fichier FAIRE
   |
   | AFFICHER  ligne
   |
Fin Pour
fichier.fermer()
```

```
fic = file(fichier,'r')
for ligne in fic:
   print ligne
fic.close()
```

Annexe B

# Les variables

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcezvous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de
variables tel que altitude, altit ou alt conviennent mieux que x pour exprimer une altitude.

**Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.**

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

— Un nom de variable est une séquence de lettres (`a` → `z` , `A` → `Z`) et de chiffres (`0` → `9`), qui doit toujours commencer par une lettre.

— Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits, à l'exception du caractère _ (souligné).

— La casse est significative (les caractères majuscules et minuscules sont distingués).
Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans tableDesMatieres, par exemple.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les 29 « mots réservés » ci-dessous (ils sont utilisés par le langage lui-même) :

```
and    assert  break   class   continue  def
del    elif    else    except  exec      finally
for    from    global  if      import    in
is     lambda  not     or      pass      print
raise  return  try     while   yield
```

Annexe C

# Formatage des chaînes de caractères

Le formatage est une technique particulièrement utile dans tous les cas où vous devez construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses.

Vous pouvez construire une chaîne formatée en assemblant deux éléments à l'aide de l'opérateur % : à gauche, vous fournissez une chaîne de formatage (un patron en quelque sorte) qui contient des **marqueurs de conversion**, et à droite (entre parenthèses) un ou plusieurs objets que Python devra insérer dans la chaîne, en lieu et place des marqueurs.

## Exemple

```
>>> coul ="verte"
>>> temp = 1.347 + 15.9
>>> print "La couleur est %s et la température vaut %s °C" % (coul,temp)
La couleur est verte et la température vaut 17.247 °C
```

Dans cet exemple, la chaîne de formatage contient deux marqueurs de conversion `%s` qui seront remplacés respectivement par les contenus des deux variables **coul** et **temp**.

Le marqueur `%s` accepte n'importe quel objet (chaîne, entier, float...).

### Afficher des nombres

Le marqueur `%d` permet d'afficher un nombre entier. Vous pouvez placer entre le % et le `d` un nombre qui indique la largeur du champ, c'est-à-dire le nombre de caractères utilisés pour afficher cet entier. S'il le faut, des espaces sont ajoutées à droite du nombre.

Les marqueurs `%f` et `%e` permettent d'afficher des réels. Vous pouvez utiliser le marqueur `%a,bf` où `a` correspond à la largeur du champ, et `b` au nombre de chiffres après la virgule

Par exemple :

```
>>> x = 1.23456
>>> print "x = %.2f" % x
x = 1.23
>>> print "x = %10.3f" %x
x =      1.235
>>> print "x = %.2e" % x
y = 1.23e+00
```

## Marqueurs de formatage

| marqueur | description |
|---|---|
| `%d` ou `%s` | entier décimal signé |
| `%o` | octal non signé |
| `%u` | décimal non signé |
| `%#x` ou `%#X` | valeur hexadécimale, préfixée respectivement par `0x` ou `0X` |
| `%x` ou `%X` | valeur hexadécimale sans préfixe |
| `%e` ou `%E` | valeur à virgule flottante, de la forme `1.23e4` ou `1.23E4` |
| `%f` ou `%F` | réel |
| `%c` | un seul caractère |
| `%s` | chaîne de caractère, mais aussi n'importe quel objet |
| `%%` | permet d'utiliser le caractère `%` dans une chaîne formatée |