Chapitre 1

Premiers éléments de langage de programmation

Sommaire

L'intégralité de ce chapitre, ou presque, est, encore une fois, extrait du document Introduction à l'algorithmique et à la programmation avec Python *de Laurent Signac*[1]

1. Disponible ici : https://deptinfo-ensip.univ-poitiers.fr

1.1 Langages

Un langage de programmation permet d'écrire un programme. Un programme, lorsqu'il est sous la forme de code source, est un texte qui exprime un algorithme (nous y reviendrons), en vue de permettre l'exécution de ce dernier sur une machine.
Les langages utilisés pour programmer sont situés quelque part entre les séquences de 0 et 1 chères à la machine et le langage naturel cher à l'humain.
Au lycée nous utiliserons deux langages : le langage dit *courant* et le langage nommé *Python*.

Le langage *courant* : c'est un langage, parfois appelé *pseudo code*, assez proche du langage naturel mais il colle tout de même aux constructions habituelles des langages de programmation (tests, boucles...).

Le langage Python : Il est généraliste, assez répandu, assez jeune, expressif et agréable à utiliser et c'est celui que les instructions ministérielles nous demandent d'utiliser!

On évoquera parfois le langage propre à certaines calculatrices (Casio et TI), même si ces langages sont voués à disparaître puisque les nouveaux modèles de ces fabricants, intègrent désormais le langage Python; c'est aussi le cas de la calculatrice Numworks qui elle est intégralement programmée en Python.

1.1.1 Le langage courant

Voici un exemple d'algorithme (suite d'instructions), destiné à convertir une température donnée en degré Farenheit (fare) en degré Celsius (celc), en langage courant :

Entrées
$fare$: nombre
Instructions
$celc \leftarrow (fare - 32)/1,8$
Sortie
$celc$

On notera la flèche $\leftarrow$ qui pourrait être remplacée par l'expression « prend la valeur » ou bien « se voit affecter la valeur ». C'est ce qu'on appelle le symbole d'affectation, ici en langage courant. Nous en reparlerons plus tard.

1.1.2 Traduction en Python

Voici ce que donnerait un tel algorithme traduit en Python :

En Python avec fonction :

```
def conversion(fare):
  celc = (fare-32)/1.8
  return celc
```

En Python sans fonction :

```
fare = input(''Entrez un entier'')
fare = int(fare)
celc = (fare-32)/1.8
print(celc)
```

Lors de la conception d'un programme un peu long, il est indispensable de le structurer en parties indépendantes. C'est le rôle de la fonction telle que donnée en exemple ci-dessus. C'est un sous-programme dans le programme qui peut être utilisée comme suit :

```
# Fonction
def conversion(fare):
   celc = (fare-32)/1.8
   return celc

# Programme principal
fare = float(input("Temperature en degres Fahrenheit "))
celc = conversion(fare)
print("En degres Celsius, cela fait ", celc)
```

Si votre bibliothèque de programmes est bien structurée, vous pourrez même utiliser des fonctions d'autres programmes ou d'autres bibliothèques dans vos programmes.
Pour une plus grande efficacité, il faut prendre l'habitude d'user et d'abuser de ces fonctions.

1.2 Algorithmes

1.2.1 À quoi sert un algorithme ?

L'algorithmique est bien plus ancienne que l'informatique, que l'ordinateur ou que le langage Python.
Les exemples les plus anciens et célèbres sont :

- les calculs d'impôts babyloniens (il y a 4 000 ans) ;
- le calcul du plus grand diviseur commun (EUCLIDE, vers −350) ;
- les premières méthodes de résolution systématique d'équations (AL KHAWARIZMI, IX^{e} siècle).

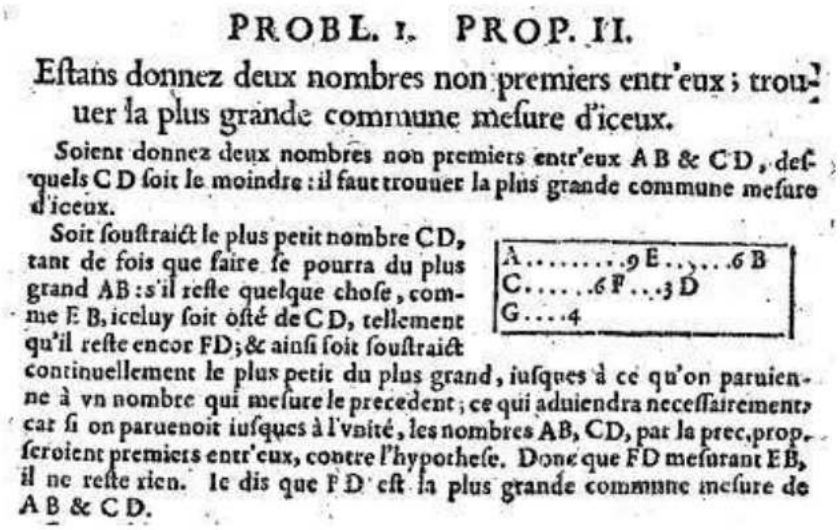
Aujourd'hui on entend par *algorithme* la réflexion préliminaire à l'écriture d'un programme d'ordinateur. C'est la partie conceptuelle de la programmation, l'abstraction d'un programme d'ordinateur.
L'algorithmique est parfois considérée comme une branche des mathématiques, parfois comme une branche de l'informatique. Les notions d'algorithme puis d'ordinateur ont été formalisées dans les années 30 et 40 par : KURT GÖDEL, ALAN TURING, JOHN VON NEUMANN ...
Les compétences en algorithmique font partie du bagage minimum d'un scientifique.

1.2.2 Un exemple célèbre : l'algorithme d'EUCLIDE

Définition 1.1. Étant donnés deux entiers, retrancher le plus petit au plus grand et recommencer jusqu'à ce que les deux nombres soient égaux. La valeur obtenue est le plus grand diviseur commun.

EXERCICE.
Appliquez l'algorithme d'Euclide, à la main, aux nombres 133 et 49.

FIGURE 1.1: Extrait de « Les Éléments » livre VII, édition de 1632



PROBL. 1. PROP. II.

Eſtans donnez deux nombres non premiers entr'eux ; trouuer la plus grande commune meſure d'iceux.

Soient donnez deux nombres non premiers entr'eux A B & C D, deſquels C D ſoit le moindre : il faut trouuer la plus grande commune meſure d'iceux.

Soit ſouſtraict le plus petit nombre C D, tant de fois que faire ſe pourra du plus grand A B : s'il reſte quelque choſe, comme E B, iceluy ſoit oſté de C D, tellement qu'il reſte encor F D ; & ainſi ſoit ſouſtraict continuellement le plus petit du plus grand, iuſques à ce qu'on paruienne à vn nombre qui meſure le precedent ; ce qui aduiendra neceſſairement, car ſi on paruenoit iuſques à l'vnité, les nombres A B, C D, par la prec. prop. ſeroient premiers entr'eux, contre l'hypotheſe. Donc que F D meſurant E B, il ne reſte rien. Ie dis que F D eſt la plus grande commune meſure de A B & C D.

A.....9 E...3...6 B
C.....6 F...3 D
G....4

Voici comment on peut procéder pour passer de l'idée à l'algorithme puis au programme, en raffinant petit à petit :

Étape 1 : Écrire proprement l'idée.
Prendre les deux nombres et, tant qu'ils ne sont pas égaux, retirer le plus petit au plus grand.

Étape 2 : Décrire plus précisément les étapes :
a et b : deux nombres
Répéter tant que a et b sont différents :
si a est le plus grand
les deux nombres deviennent $a-b$ et b
sinon (lorsque b est le plus grand)
les deux nombres deviennent a et $b-a$
le pgcd est a

Étape 3 : Écrire en langage courant ou pseudo code :
fonction pgcd(a, b : entiers)
Répéter tant que $a \neq b$:
si $a > b$
$a \leftarrow a - b$
sinon
$b \leftarrow b - a$
retourner a

Étape 4 : Écrire en langage de programmation (ici Python) :

```
def pgcd(a,b):
    while a != b:
        if a > b:
            a = a-b
        else:
            b = b-a
    return a
```

EXERCICE.
Entrer cette fonction dans votre calculatrice Numworks en Python et entrer print(pcgd(133,49)) dans la console pour vérifier ce qu'elle renvoie.

1.3 Types, affectations, variables

1.3.1 Types simples

Le type d'une donnée caractérise sa nature : entier, nombre à virgule, couleur, liste, etc. et par voie de conséquence les opérations qu'il est possible de lui appliquer (on peut ajouter des nombres, pas forcément des couleurs).
Les types disponibles dans un langage informatique dépendent fortement du codage utilisé pour représenter les données. En algorithmique, on peut utiliser des types « mathématiques » qui n'existent pas dans les machines (les réels par exemple).
Quelques exemples de types simples en Python :

Nom commun	Python	Remarques
Entiers	int	Voir le paragraphe ci-dessous
Réels	float	Les calculs en flottants sont approchés
Booléen	bool	Vrai ou faux
Caractère		En Python, il n'y a pas de type caractère

Cas des entiers et de réels

Les entiers, par exemple, ont une taille limitée en Python en fonction de la mémoire disponible (l'entier maximum est considérablement grand). L'ensemble des entiers « mathématiques », lui, n'est pas limité.
Les réels que l'on manipule parfois en mathématique n'existent pas en Python où ils sont remplacés par des nombres à virgule flottante (ou flottants). En conséquence, tous les calculs sur ordinateur utlisant les flottants sont par nature faux (ou approchés).
Le type *int*, permet de représenter les entiers de taille machine (quelques octets) ou les grands entiers (limités uniquement par la taille de la mémoire). Le passage de l'un Ã l'autre (entiers machines ou grands entiers) est transparent pour l'utilisateur (depuis la version 3 de Python).
Pour indiquer une valeur entière, il est possible d'utiliser différentes bases :

```
>>> 42 # en decimal
>>> 0o52 # en octal
>>> 0x2A # en hexadecimal
>>> 0b101010 # en binaire
```

Les opérations disponibles sur les nombres entiers sont résumées dans la table 1.1 page suivante.

1.3.2 Variables

Il faut voir *celc* comme une variable, c'est-à-dire une case prévue pour recevoir un contenu.
Une variable posède plusieurs caractéristiques :

identificateur (nom) Il doit être bien choisi, surtout dans un programme long. Il ne comporte aucune espace ou carcatère spécial et ne peut pas être un mot clé. Chaque langage dispose de règles très précises décrivant la syntaxe des identificateurs

type Dans certains langages, comme Python, le type d'une variable peut changer en cours d'éxecution du programme. Dans d'autres il est fixé une fois pour toute.

valeur C'est l'objet / la valeur désigné par la variable

portée On ne peut utiliser le nom d'une variable que dans la fonction qui la contient.

TABLE 1.1: Opérations sur les entiers

Opération	Type du retour	Description
x + y	int	Somme
x - y	int	Différence
x * y	int	Produit
x / y	float	Quotient
x // y	int	Quotient de la division euclidienne
x % y	int	Reste de la division euclidienne
-x	int	Opposé de x
abs(x)	int	Valeur absolue de x
float(x)	float	Conversion vers un float
int(x)	int	Conversion vers un int
str(x)	str	Conversion en chaîne de caractères
hex(x)	str	Représentation hexadécimale
oct(x)	str	Représentation octale
bin(x)	str	Représentation binaire

1.3.3 Affectation

L'instruction $celc \leftarrow (fare - 32)/1,8$ s'effectue de la manière suivante : l'algorithme (ou la machine si cet algorithme est implémenté dans une machine) évalue ou calcule la valeur de $(fare - 32)/1,8$ puis stocke cette valeur dans la variable nommée $celc$.
La flèche $\leftarrow$ est le symbole en langage courant pour indiquer l'affectation. Ce symbole varie selon les langages de programmation. En Python ce symbole est le signe égal =
On écrit alors :

```
celc = (fare - 32)/1.8
```

Remarque. Sur les calculatrices Casio et TI, dans leur langage de programmation particulier, l'affectation se fait dans le sens inhabituel des autres langages de programmation :$(f - 32)/1,8 \rightarrow c$ par exemple.

1.3.4 Entrées et sorties standards

Vos programmes peuvent permettre la saisie au clavier avec la fonction *input* :

```
>>> a = input('Entrez votre nom : ')
Entrez votre nom : Bond
>>> print(a, type(a))
Bond <class 'str'>
```

La fonction *input* renvoie une chaîne de caractères, mais on peut la convertir au passage :

```
>>> a = int(input('Entrez votre Ã¢ge : '))
Entrez votre age : 3
>>> print(a, type(a))
3 <class 'int'>
```

1.4 Fonctions et procédures

Les fonctions et procédures sont la pierre angulaire de la programmation procédurale. Il y aura deux points fondamentaux à retenir dès la fin de lecture de cette section :

- comprendre pourquoi on écrit des fonctions ;
- s'obliger à le faire dès les premiers programmes.

Commençons par un exemple en Python :

```
def calculttc(val, taux):
    """
    Calcule le montant ttc, connaissant la somme
    hors taxe (val) et le taux (par ex 19.6)
    """
    ttc=val * (1 + taux / 100)
    return ttc
```

1.4.1 Instruction *return*

Dans tous les langages algorithmiques, y compris Python, l'instruction retourner (*return*) fait se terminer la fonction ou la procédure (dans le cas d'une procédure, *return* n'est pas suivi d'une valeur), même si cette instruction est exécutée avant la fin du texte de la fonction.
Une fois la fonction *calculttc* lue par l'interpréteur, il est possible de l'utiliser ainsi :

```
>>> calculttc(1000, 19.6)
=> 1196
```

Notons qu'il convient de bien séparer la définition et l'utilisation de la fonction :

définition : on indique comment « marche »la fonction et comment on s'en sert (déclaration). La fonction n'est pas utilisée (exécutée), mais juste « lue » pour être connue de l'interpréteur.

utilisation : on exécute de manière effective le code de la fonction pour des valeurs d'entrées fixées (1 000 et 19,6 dans l'exemple).

Voici quelques occasions dans lesquelles vous devez écrire des fonctions (ou des procédures) :

1. Le bloc de programme que vous écrivez ne tient pas entier à la vue sur votre écran.
2. Vous utilisez plus d'une fois des portions de code exactement identiques, ou presque identiques. Vous gagnerez en clarté et en maintenabilité à écrire une fonction à la place de ce bloc et à l'appeler plusieurs fois.
3. Vous ne savez pas exactement si la méthode de résolution employée pour telle tâche est la meilleure. Mettez-la dans une fonction et utilisez cette fonction. Si vous trouvez une meilleure façon de procéder, il suffira de modifier la fonction sans toucher au reste.
4. Le problème que vous traitez est difficile : découpez-le en fonctions que vous écrirez et testerez séparément les unes des autres (le bénéfice que vous aurez à pouvoir tester très facilement des portions de programme est énorme). Dans chaque fonction indépendante, vous pourrez choisir des noms de variables appropriés, qui rendront votre code plus simple à comprendre.

1.4.2 Fonction ou procédure ?

Il ne faut pas confondre procédure et fonction, même si leur définition, en Python par exemple, est similaire :

- Une fonction calcule. Elle vaut quelque chose (exemples : pgdc et calculttc)
- Une procédure n'a pas de valeur, mais elle fait/modifie quelque chose (exemple : afficher à l'écran, modifier un objet). On dit qu'elle a un effet de bord (elle modifie quelque chose qui est en dehors de la procédure).

Même si la différence fonction/procédure n'est pas syntaxique en Python, elle est très importante :

```
# Ceci est une fonction
def foncttc(val, taux) :
    ttc = val * (1 + taux / 100)
    return ttc
```

```
# Ceci est une procedure
def procttc(val, taux) :
    ttc = val * (1 + taux / 100)
    print(ttc)
```

On pourrait penser que la fonction *foncttc* et la procédure *procttc* sont équivalentes. Ce serait une erreur. La fonction est plus générale. On ne peut pas écrire par exemple :

```
   # La ligne suivante ne fonctionne pas
>>> print("Double du prix ttc : ", procttc(100, 19.6) * 2)
```

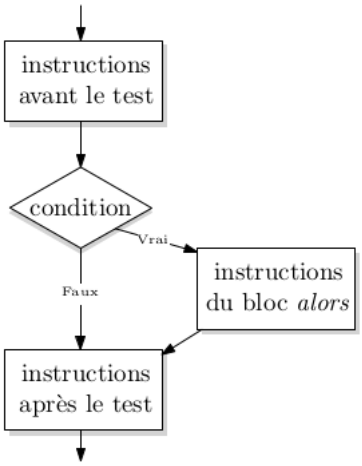
En revanche, la même ligne, en utilisant *foncttc* fournirait le bon résultat. Assurez-vous d'avoir bien compris pourquoi la ligne qui précède marche avec une fonction et pas avec une procédure. C'est vraiment très important.

1.4.3 À retenir

1. Une fonction *vaut* quelque chose, une procédure *fait* quelque chose.
2. La déclaration d'une fonction suffit (normalement) à savoir ce qu'elle calcule et comment on l'utilise, mais pas comment elle le calcule.
3. On écrit des fonctions et des procédures pour ne plus avoir à se soucier de la façon de régler le problème qu'elles traitent. En conséquence, être obligé de retoucher le code d'une fonction pour un problème particulier indique généralement que la fonction a été mal conçue au départ.
4. Généralement, lorsqu'on a terminé l'écriture d'une fonction et qu'on l'a testée, on souhaite ne plus avoir à revenir dessus et ne plus devoir la modifier.

1.5 Conditions

Les conditions en algorithmique sont traduites en langage courant par les opérateurs « si … alors … » ou « si … alors … sinon … » qui fonctionnent selon les schémas ci-dessous (avec un exemple fourni en Python) :

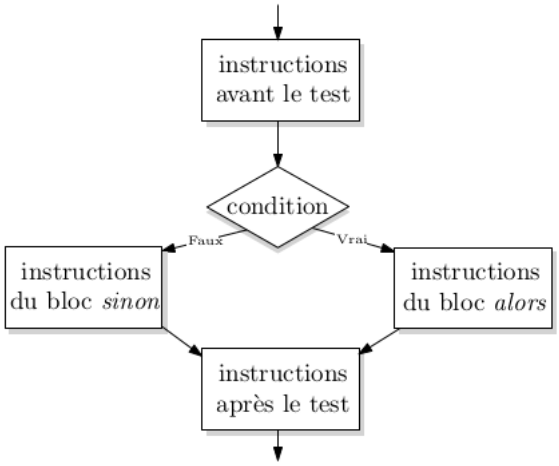


Principe du fonctionnement :

```
<instructions avant le test>
if <condition>:
    <instructions du bloc 'alors'>
<instructions apres le test>
```

Exemple.

```
def divisible(n,e):
    if n % e == 0:
        print(n, 'est divisible par', e)
        print('Bravo', n)
    print('Merci au-revoir')
```



Principe du fonctionnement :

```
<instructions avant le test>
if <condition>:
    <instructions du bloc 'alors'>
else:
    <instructions du bloc 'sinon'>
<instructions apres le test>
```

Exemple.

```
def divisibleoupas(n,e):
    if n % e == 0:
        print(n, 'est divisible par', e)
    else:
        print(n, 'n'est pas divisible par', e
    print('Merci au-revoir')
```

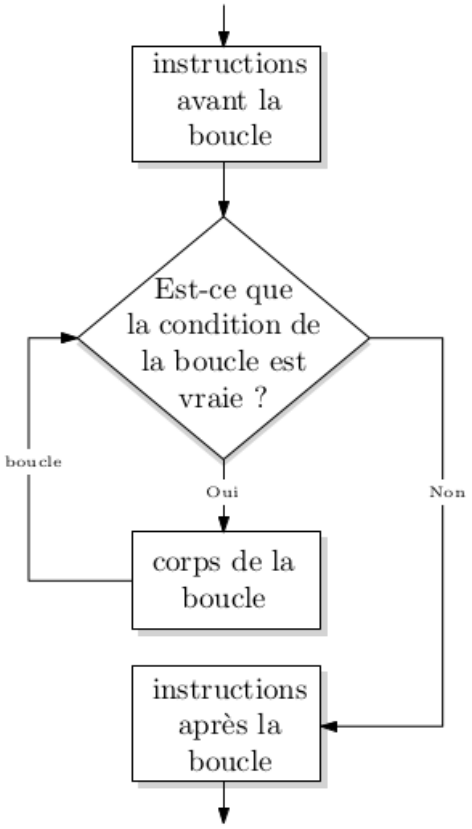
Remarques.

- En langage Python, « $n \% e$ » renvoie le reste de la division de n par e.
- Le double signe égal « == » est l'équivalent du signe égal en mathématiques.
- $a == b$ est un booléen qui prend la valeur *vrai* (*true*) si a est égal à b et la valeur *faux* (*false*) si a n'est pas égal à b.

EXERCICE.
Identifiez chaque partie des programmes qui précèdent avec les blocs correspondants de l'organigramme.

1.6 Boucles

1.6.1 Boucles « Tant que »



Principe du fonctionnement :

```
<instructions avant le test>
while <condition>:
    <corps de la boucle>
<instructions apres la boucle>
```

Exemple. Nous souhaitons faire un programme de jeu qui propose à l'utilisateur de deviner un nombre entre 1 et 100 en faisant des propositions. À chaque proposition, le programme lui iniduqe si le nombre proposé est trop petit ou trop grand. Le jeu s'arrête lorsque le nombre est découvert.

```
import random
nb = random.randint(1,100)
ch = -1
while nb != ch :
    ch = int(input('Entrez un nombre'))
    if ch < nb : print('Trop petit')
    if ch > nb : print('Trop grand')
print('Bravo')
```

EXERCICE.
Identifiez chaque partie du programme qui précède avec les blocs correspondants de l'organigramme.

1.6.2 Boucles Répéter Pour et itérateurs

Beaucoup de langages offrent un autre type de boucle, permettant de gérer un compteur. Voici un exemple qui affiche des tables de multiplication :

```
Repeter Pour i allant de 1 a 10 :
    Repeter Pour j allant de 1 a 10 :
        Ecrire(i,'x',j,'=',i*j)
```

Python n'est pas en reste pour ce type de boucle, même si la construction offerte est plus générale que la simple boucle avec compteur.
La traduction de l'algorithme qui précède en Python donnerait :

```
for i in range(1,11):
    for j in range(1,11):
        print(i,'x',j,'=',i*j)
```

La boucle *for* de Python permet en fait d'itérer sur n'importe quel objet itérable. Or l'objet *range*(a,b) est itérable et « contient » les entiers de a à $b-1$ inclus. C'est pourquoi dans l'exemple qui précède i prend successivement toutes les valeurs de 1 à 10.
Avec la boucle *for* de Python, il est possible de traverser toutes sortes d'objets, par exemple :

```
for lettre in 'Hello World' :
    print(lettre)
```

1.7 Exercices

Les exercices seront puisés dans le manuel (de mathématiques).