

## Chapitre 1 : Introduction à la programmation – Le Robot.

### 1. Introduction

L'ordinateur effectue des traitements sur des données. L'objectif de cette partie est de se sensibiliser et se familiariser avec la logique sous-jacente à l'activité de l'ordinateur : la logique des traitements. On montre à l'aide d'une interface simple, le maniement d'un robot, comment décomposer un problème en problèmes plus simples et les structures de contrôle du déroulement d'un traitement.

### 2. Présentation du robot

#### 2.1. L'environnement

Un robot se déplace dans un domaine rectangulaire de dimensions finies, divisé en cases (Fig 1). Cet espace sera délimité par une frontière. Les cases pourront être vides, marquées ou contenir un trésor ou un mur. C'est le robot qui dépose éventuellement des marques pour laisser la trace de son passage.

Les cases sont numérotées à partir de 0, le coin supérieur gauche étant à la position (0, 0). Ici la grille fait 7x5, les index des cases horizontalement vont de 0 à 6 et verticalement de 0 à 4. Une marque est positionnée en (0,1) le trésor en (4,2) et un mur est positionné de (2,2) jusque (2,3). Le robot se trouve en (1,1) orienté vers la droite donc vers l'est.

Par convention le haut sera le Nord, le bas le Sud, la droite sera l'Est et la gauche l'Ouest.



Fig 1 : Le Robot

```

76 *exo4.py - C:/WINDOWS/Profiles/al...
File Edit Format Robot Help
def init():
    setDelai(0.2)
    initTresor(ALEA)
    initRobot(ALEA, EST)
def main():
    init()
    deposerMarque()
    avancer()
    avancer()
    avancer()
    deposerMarque()
Ln: 11 Col: 19

```

Fig 2 : La fenêtre de code



Fig 3 : Le browser

Le robot sait réaliser quelques actions élémentaires (2.3) et faire quelques observations sur son environnement, appelés des tests (2.4).

Quelques opérations d'initialisation et d'interaction avec l'utilisateur sont ajoutées à l'environnement (2.2).

Le code des programmes s'écrit directement en Python (Fig 2) et une analyse syntaxique est effectuée au moment de l'exécution. La figure 4 montre une erreur de syntaxe détectée.

Un browser (fig 3) affiche les méthodes du programme trouvées lors de la dernière écriture sur disque.

```

def init():
    setDelai(0.2)
    initTresor(ALEA)
    initRobot(ALEA, EST)
def main():
    init()
    deposerMarque()
    avancer()
    avancer()
    avancer()
    deposerMarque()

```

Fig 4 : Une erreur de syntaxe

## 2.2. Les méthodes d'initialisation

Si vous ne précisez rien, le robot et le trésor sont placés au hasard sur le terrain, la direction du robot est aléatoire. Vous pouvez imposer (dans certaines limites) une position et une orientation initiale. Attention : vous ne pouvez utiliser ces instructions qu'une seule fois par programme ; vous ne pouvez pas les utiliser après une quelconque action du robot. Vous pouvez également positionner des murs d'une ou plusieurs cases sur le terrain

### 2.2.1. Initialisation du robot

- Place du robot :
  - au hasard : HASARD, sur un bord : BORD, pas sur un bord : PAS\_BORD, un coin : COIN
  - bord nord : BO, bord sud : BS, bord est : BE, bord ouest : BO
  - coin nord est : CNE, coin nord ouest : CNO, coin sud est : CSE, coin sud ouest : CSO
- Orientation du robot : HASARD, NORD, SUD, EST et OUEST.
- Méthodes:
  - `initRobot(position, orientation)` permet de positionner le robot. Par exemple pour placer le robot dans le coin sud ouest orienté nord on écrira `initRobot(CSO, NORD)`.
  - `placeRobot(x, y)` le place à une position en (x, y) : `placeRobot(2, 3)`

### 2.2.2. Initialisation du trésor

- Place du trésor : les mêmes que pour le robot
- Méthodes:
  - `initTresor(position)` positionne le trésor : `initTresor(BORD)`
  - `placeTresor(x, y)` le place à une position en (x, y) : `placeTresor(2, 3)`

### 2.2.3. Initialisation des murs

- Méthodes:
  - `placeMur(x, y)` : place un mur d'une case en (x, y) : `placeMur(2, 3)`
  - `placeMurH(x, y, w)` : place un mur horizontal de w cases en (x, y) : `placeMurH(2, 3, 3)`
  - `placeMurV(x, y, h)` : place un mur vertical de h cases en (x, y) : `placeMurV(2, 3, 4)`

*remarque* : SI l'on essaie de placer un mur sur la case où est situé le robot, le mur n'est pas placé sans message d'erreur.

### 2.3. Les actions

- **avancer()** : Le robot avance dans sa direction d'une case. S'il se trouvait devant un mur, le programme s'arrête et vous invective (en principe pas trop méchamment).
- **droite()** : Le robot tourne d'un quart de tour vers la droite (ou, si vous préférez, dans le sens des aiguilles d'une montre, ou encore, dans le sens anti-trigonométrique).
- **deposerMarque()** : Le robot dépose une marque à l'endroit où il se trouve. Des marques empilées ne correspondent qu'à une seule marque.
- **enleverMarque()** : Le robot efface la marque à l'endroit où il se trouve. Si cet emplacement ne contient pas de marque, cette action ne fait rien.

### 2.4. Les test

- **devantMur()** : la réponse est oui s'il se trouve devant un mur, non dans tous les autres cas.
- **devantTresor()** : la réponse est oui s'il se trouve devant le trésor, non dans tous les autres cas.
- **devantMarque()** : la réponse est oui s'il se trouve devant une marque, non dans tous les autres cas.
- **surMarque()** : la réponse est oui s'il se trouve sur une marque, non dans tous les autres cas.

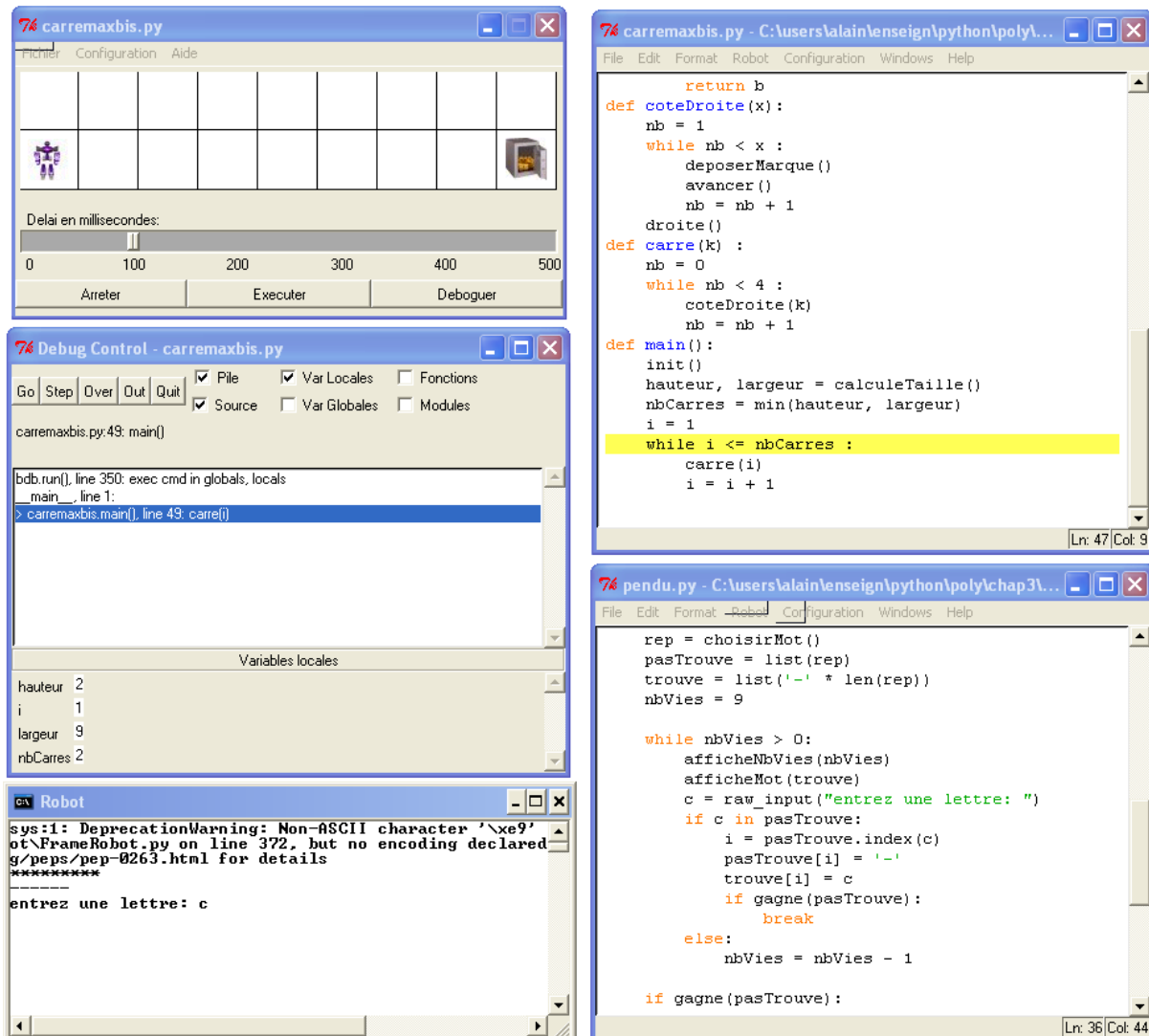
Les tests peuvent être combinés avec des opérateurs logiques :

- Le Non « **not** » inverse la valeur du test. Par exemple **not devantMur()** est vrai si le robot n'est pas devant un mur.
- Le Et « **and** » permet de tester deux conditions. Par exemple **devantMur() and surMarque()** est vrai si le robot est sur une marque et devant un mur. Ce test est faux dans tous les autres cas, c'est-à-dire, si le robot n'est pas devant un mur ou si le robot n'est pas sur une marque.
- Le Ou « **or** » permet de tester si au moins une conditions est vraie. Par exemple **devantMur() or surMarque()** est vrai si le robot est sur une marque et devant un mur. Ce test est faux dans tous les autres cas, c'est-à-dire est vrai si le robot est sur une marque ou devant un mur. Ce test est faux dans tous les autres cas, c'est-à-dire, si le robot n'est pas devant un mur et si le robot n'est pas sur une marque.

### 2.5. Compléments : méthodes d'interaction avec l'utilisateur

- Si vous le désirez vous pouvez introduire une pause dans votre programme, par exemple à la fin de l'initialisation, grâce à la méthode **pause(tps)**. **pause(1)** introduit une pause de une seconde pendant laquelle le robot ne fait rien.
- **largeurGrille()** renvoie la largeur de la grille, par exemple 7 pour la figure 1.
- **hauteurGrille()** renvoie la hauteur de la grille, par exemple 5 pour la figure 1.
- **choisir(x, y)** retourne un entier entre x inclus et y exclus.

### 3. Debugger et Terminal



Le **robot** possède un débogueur qui permet d'exécuter les instructions pas à pas. On l'active (fenêtre Control Debug) en cliquant sur le bouton Deboguer ou dans le menu Robot → Deboguer de l'éditeur. On peut alors placer ou enlever des points d'arrêt (ligne 47 de `carremaxbis.py`) en cliquant avec le bouton droit de la souris. Les commandes du débogueur sont les suivantes :

- **Go** : exécute le programme et s'arrête au premier point d'arrêt s'il y en a un.
- **Step** : exécute une seule ligne d'instruction puis attend une commande de l'utilisateur.
- **Over** : exécute une procédure sans entrer dans son code.
- **Out** : exécute le code restant d'une fonction et s'arrête à la sortie de cette fonction.
- **Quit** : Arrête l'exécution du programme et sort du débogueur.

Les cases à cocher permettent de visualiser les variables et leurs valeurs.

Le **terminal** (en bas à gauche) permet l'affichage de texte (instruction `print`) et la lecture de données (instruction `input`), ici dans le jeu du pendu (en bas à droite) le programme demande à l'utilisateur de proposer une lettre ligne 36: `c = raw_input("entrez une lettre: ")`, l'utilisateur a entré la lettre 'c'.

## 4. Les structures de contrôle

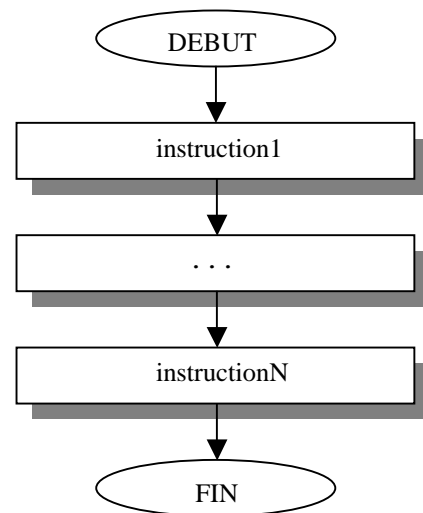
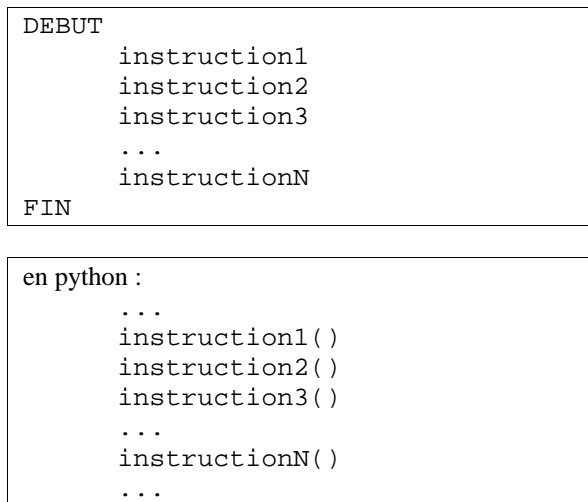
Un programme est une suite d'actions. L'exécution du programme correspond à l'exécution de la suite des actions qui le compose. Les structures de contrôle indiqueront comment s'organisent les actions dans le temps, comment elles s'enchaînent.

Il y a 3 types de structures : la séquence, l'alternative et l'itérative.

### 4.1. La séquence

#### 4.1.1. Algorithmique

La séquence est la structure la plus simple que l'on puisse trouver. L'exécution d'une telle structure correspond à l'exécution des instructions les unes à la suite des autres. En langage algorithmique, cela donne :



On exécute d'abord `instruction1` puis `instruction2`, `instruction3`,... et enfin `instructionN`.

Quelques remarques sur le code python :

- Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (le point-virgule en java). En Python, c'est le caractère de saut à la ligne qui joue ce rôle. On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial `#`. Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré.
- Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques. En *Java* par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes difficiles à relire. On conseille donc à tous les programmeurs qui utilisent ces langages de se servir *aussi* des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.
- Avec Python, *vous devez* utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

### 4.1.2. Exemple

- Le robot se trouve dans le coin sud-ouest, orienté vers le nord. Avancer de trois cases.

```
#le programme principal
def main():
    #initialisation
    initRobot(CSO, NORD) #on place le robot
    #fait avancer le robot de 3 cases
    avancer()
    avancer()
    avancer()
```

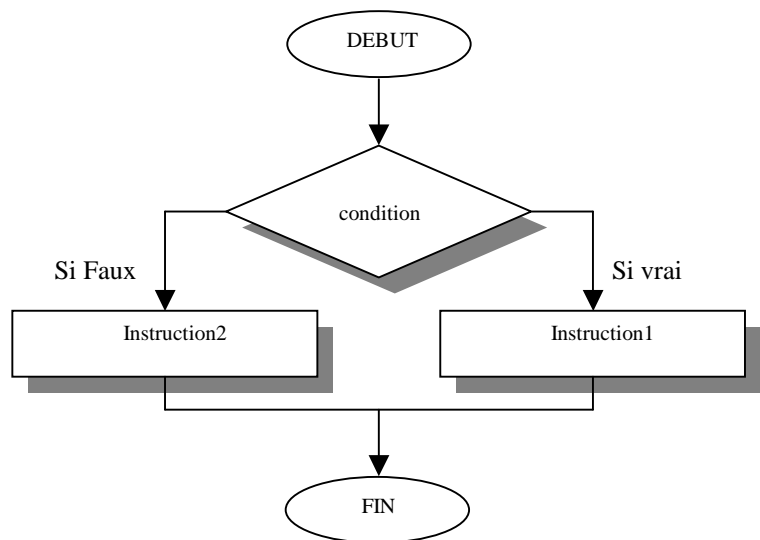
## 4.2. L'alternative

### 4.2.1. Algorithmique

L'alternative est une structure qui permet l'exécution d'une action ou d'une séquence d'actions à partir du moment où une condition est vérifiée.

```
DEBUT
    SI condition FAIRE
        instruction1
        ...
    SINON
        instruction2
        ...
    FIN DU SI
FIN
```

```
en python :
...
if condition():
    instruction1()
    ...
else:
    instruction2()
    ...
...
```



Si la condition `condition` est vraie, les instructions dans `instruction1` sont exécutées. Dans le cas contraire, ce sont les instructions dans `instruction2` qui sont exécutées. La partie `SINON` n'est pas obligatoire, quand elle n'existe pas et que la condition `condition` n'est pas vérifiée, aucun traitement n'est réalisé.

Les blocs d'instruction situés dans les parties `if` et `else` doivent être indentés exactement au même niveau (comptez un décalage de 4 caractères, par exemple).

## 4.2.2. Exemples

- Le robot se trouve dans le coin sud-ouest, orienté au hasard, il avance si la case devant lui est libre, sinon il fait demi-tour.

```
def main():
    #initialisation
    initRobot(CSO, HASARD)
    if not devantMur():
        avancer()
    else:
        droite()
        droite()
```

```
def main():
    #initialisation
    initRobot(CSO, HASARD)
    if devantMur():
        droite ()
        droite ()
    else:
        avancer()
```

- Le robot se trouve sur un bord, orienté au hasard, il avance si la case devant lui est libre, sinon il ne fait rien.

```
def main():
    initRobot(BORD, HASARD)
    if not devantMur():
        avancer()
```

```
def main():
    initRobot(BORD, HASARD)
    if devantMur():
        pass
    else:
        avancer()
```

Notez ici l'utilisation de l'instruction `pass` qui signifie ne rien faire, en effet la présence d'une instruction est obligatoire après le `if`

## 4.2.3. Si imbriqués

Les alternatives peuvent parfois imbriquées, il faut bien sûr respecter l'indentation des différents blocs.

```
DEBUT
    SI condition1 FAIRE
        instruction1
    SI condition2 FAIRE
        instruction2
    SINON
        instruction3
    FIN DU SI
    SINON
        instruction4
    FIN DU SI
FIN
```

```
en python :
...
    if condition1():
        instruction1()
        if condition2():
            instruction2()
        else:
            instruction3()
    else:
        instruction4()
...
```

- Exemple :

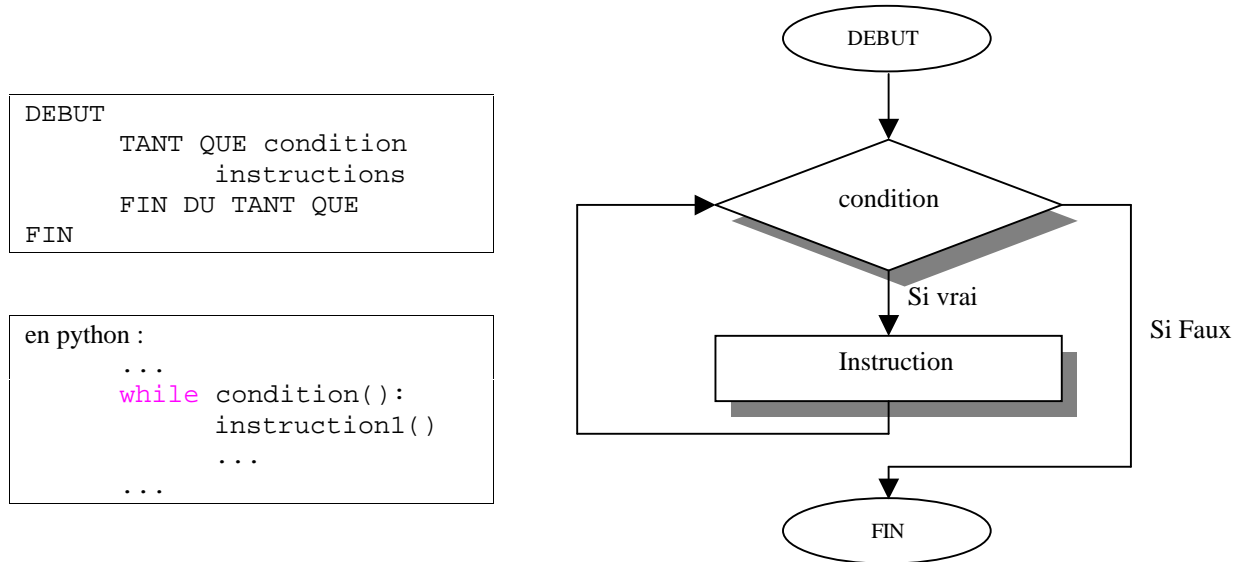
Le robot se trouve dans le coin sud-ouest, orienté au hasard, s'il se trouve devant le mur il ne fait rien sinon il avance d'une case, tourne à droite puis avance d'une case si c'est possible.

```
def main():
    initRobot(BORD, HASARD)
    if not devantMur():
        avancer()
        droite()
        if not devantMur():
            avancer()
```

## 4.3. L'itération

### 4.3.1. Algorithmique

L'itération est une structure qui permet l'exécution d'une action ou d'une séquence d'actions tant qu'une condition est vérifiée.



Si la condition `condition` est vraie, les instructions dans `Instructions` sont exécutées. Puis, on retourne tester la condition. Dans le cas contraire, l'itération est terminée. On dit alors que l'on sort de la boucle. La condition est toujours évaluée avant de faire le traitement. Donc, si la condition n'est jamais vraie, le traitement n'est jamais effectué.

### 4.3.2. Exemple

Le robot se trouve dans le coin sud-ouest, orienté vers le nord. Le trésor se trouve sur le bord ouest mais pas dans le coin sud-ouest. Dirigez le robot pour qu'il atteigne le trésor.

```
def main():
    initTresor(BO)
    initRobot(CSO, NORD)
    while not devantTresor():
        avancer()
    avancer()
```

### 4.3.3. Structures de contrôle imbriqués

Les structures de contrôle peuvent bien sûr être imbriquées comme au paragraphe 3.2.3. On peut trouver un SI dans un SI, un TANT QUE dans un SI, un SI dans un TANT QUE etc ...

## 4.4. Les procédures

### 4.4.1. Algorithmique

L'écriture de procédures a principalement deux avantages. Elle permet de rendre les algorithmes plus lisibles, et donc plus faciles à comprendre, à corriger, à modifier ; elle permet de << factoriser >> les algorithmes et donc de rendre plus courte leur écriture : un morceau d'algorithme qui se répète souvent peut devenir une procédure qui sera utilisée chaque fois que cela sera nécessaire.

Par exemple, on veut faire faire le tour du terrain au robot partant du coin Nord Ouest, orienté vers l'est. Il suffit d'écrire l'algorithme suivant :

```

DEBUT
    Tant Que Non Devant Mur
        Avancer
    Fin du Tant que
    A droite
    Tant Que Non Devant Mur
        Avancer
    Fin du Tant que
    A droite
    Tant Que Non Devant Mur
        Avancer
    Fin du Tant que
    A droite
    Tant Que Non Devant Mur
        Avancer
    Fin du Tant que
    A droite
FIN

```

```

def main():

    initTresor(HASARD)
    initRobot(CNO, EST)

    while not devantMur():
        avancer()
    droite()
    while not devantMur():
        avancer()
    droite()
    while not devantMur():
        avancer()
    droite()
    while not devantMur():
        avancer()
    droite()

```

On peut créer une procédure qui dit : « Je vais jusqu'au mur et je tourne à droite »  
Appelons Jusqu'au mur droite cette procédure. Elle s'écrit facilement

```

Procédure Jusqu'au mur droite
DEBUT
    Tant Que Non Devant Mur
        Avancer
    Fin du Tant que
    A droite
FIN

```

```

def jusquAuMurDroite():
    while not devantMur():
        avancer()
    droite()

```

Maintenant, le programme va utiliser la procédure en s'écrivant plus simplement, on en profite ici pour placer l'initialisation dans une procédure init :

```

DEBUT
    Jusqu'au mur droite
    Jusqu'au mur droite
    Jusqu'au mur droite
    Jusqu'au mur droite
FIN

```

```

def init():
    initTresor(HASARD)
    initRobot(CNO, EST)
def jusquAuMurDroite():
    while not devantMur():
        avancer()
    droite()
def main():
    init()
    jusquAuMurDroite()
    jusquAuMurDroite()
    jusquAuMurDroite()
    jusquAuMurDroite()

```

## 4.4.2. Exemple

Le robot se trouve sur le bord sud, orienté vers le nord. Un mur de 3 cases de large est placé au dessus de lui sur n'importe quelle ligne mais pas dans la 1<sup>o</sup> colonne. Faire avancer le robot jusqu'au bord nord en marquant les cases (fig 9).

Principe : Faire avancer le robot jusqu'au mur, lui faire contourner par l'ouest (la gauche) puis faire de nouveau avancer le robot jusqu'au mur (fig 10).

```
def init():
    initRobot(BS, NORD)
    placeMurH(2, 2, 3)

def gauche():
    droite()
    droite()
    droite()

def jusquAuMur():
    while not devantMur():
        deposerMarque()
        avancer()

def chercheSortie():
    while devantMur():
        gauche()
        deposerMarque()
        avancer()
        droite()

def main():
    init()
    jusquAuMur()
    chercheSortie()
    jusquAuMur()
```

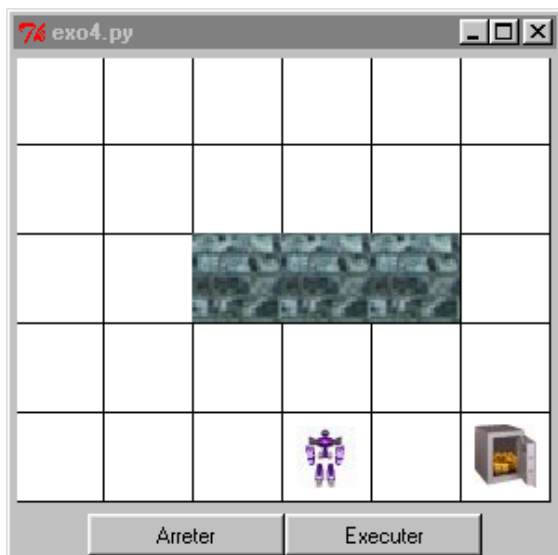


fig 9 : initialisation

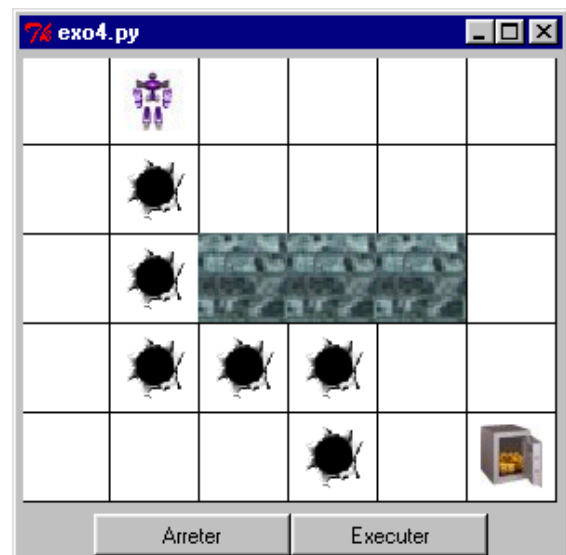


fig 10 : résultat

## Chapitre 2 : Variables et fonctions.

### 1. Variables, procédures et fonctions.

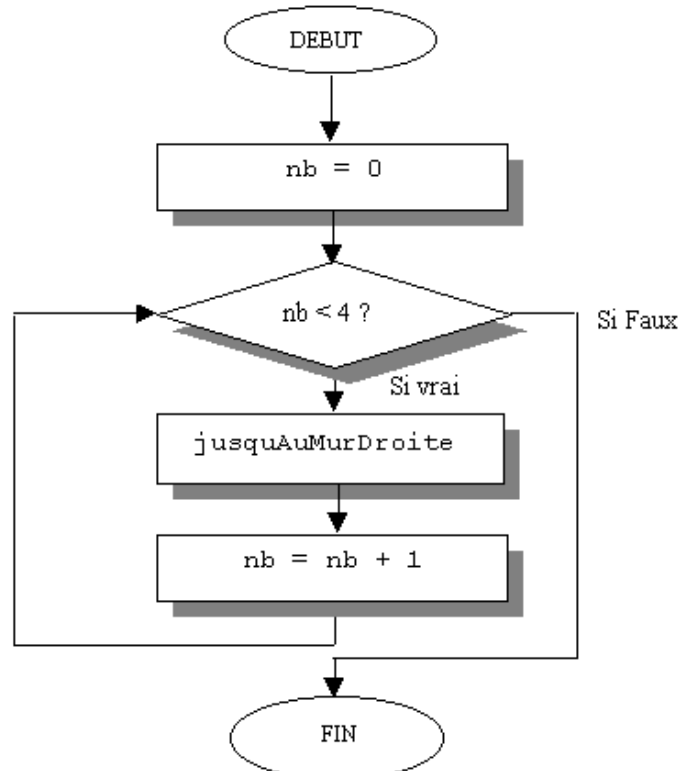
#### 1.1. Introduction aux variables.

Dans le premier cours nous avons comment écrire un programme et l'intérêt de décomposer un programme en procédures. Dans cette partie nous allons introduire les notions de variables et de fonctions.

Reprenons l'exercice 3.4.1 le robot est dans le coin Nord Ouest, orienté vers l'est et doit faire le tour du terrain. Nous avons écrit :

```
def init():
    initRobot(CNO, EST)
def jusquAuMurDroite():
    while not devantMur():
        avancer()
    droite()
def main():
    init()
    jusquAuMurDroite()
    jusquAuMurDroite()
    jusquAuMurDroite()
    jusquAuMurDroite()
```

Il serait plus intéressant d'écrire que le robot doit effectuer 4 fois la procédure `jusquAuMurDroite`. Pour cela nous allons définir une variable qui contiendra le nombre de côtés parcouru sur le terrain. Appelons cette variable `nb`. Au départ `nb` est égal à 0, lorsque le premier côté est parcouru `nb` est égal à l'ancienne valeur (0)+1 soit : 1, le robot n'a pas terminé puisqu'il doit parcourir 4 côtés. Il parcourt le 2<sup>o</sup> côté et `nb` vaut maintenant (1)+1 soit : 2, puis le 3<sup>o</sup> et enfin le 4<sup>o</sup>. La variable `nb` vaut maintenant (3)+1 soit : 4. Les 4 côtés sont parcourus et le robot a terminé. Autrement dit tant que `nb` est strictement inférieur à 4 le robot doit parcourir un côté. Nous pouvons représenter l'organigramme du programme :



Le programme devient (on ne réécrira que la procédure main) :

```
def main():
    init()
    nb = 0
    while nb < 4 :
        jusquAuMurDroite()
        nb = nb + 1
```

ou en décomposant le programme :

```
def tourTerrain():
    nb = 0
    while nb < 4 :
        jusquAuMurDroite()
        nb = nb + 1
def main():
    init()
    tourTerrain()
```

Quelques remarques :

- La ligne `nb = 0` est appelée une *affectation* : on affecte la valeur 0 à la variable `nb`, `nb` représente une case mémoire où seront stockées les différentes valeurs de `nb` au cours du programme.
- La ligne `nb = nb + 1` n'est pas une égalité comme en mathématiques mais une affectation : on calcule d'abord la valeur de `nb + 1` puis on copie cette valeur dans la case mémoire référencée par `nb`. L'ancienne valeur est perdue.

## 1.2. Procédures avec paramètres.

Puisque le robot sait maintenant faire le tour du terrain il doit être facile de modifier la procédure `tourTerrain` pour lui faire effectuer un carré de 4x4 ou de 6x6. Il suffit de conserver la même structure du programme en remplaçant la procédure `jusquAuMurDroite` par la procédure `avancerDroite` qui fait avancer de 3 ou 5 cases suivant le cas. On écrira `avancerDroite(3)` pour faire avancer le robot de 3 cases ou `avancerDroite(5)` pour le faire avancer de 5 cases. Puisqu'au moment de l'écriture de la procédure le nombre de cases dont il faut faire avancer le robot n'est pas connu nous lui affecterons un nom, par exemple `x`.

```
def avancerDroite(x):
    nb = 0
    while nb < x :
        avancer()
        nb = nb + 1
    droite()
```

Comment faire un carré de `k` cases ?

```
def carre(k):
    nb = 0
    while nb < 4 :
        avancerDroite(k-1)
        nb = nb + 1
```

En appelant `carre(4)` `k` vaudra 4 dans le corps de la procédure `carre`. Celle-ci appelle 4 fois la procédure `avancerDroite(3)`, donc dans le corps de `avancerDroite` `x` vaudra la valeur passée en paramètre c'est à dire 3. On peut maintenant écrire :

```
def main():
    init()
    carre(4)
    carre(6)
```

### 1.3. Fonctions.

Jusqu'à maintenant nous avons défini des procédures avec ou sans paramètres. Rappelons qu'une procédure *réalise* une action. Dans le cadre du robot ce sera "avancer jusqu'au mur" etc ... Nous avons également vu qu'il existait des *tests*. Ces tests retournent une valeur puisqu'ils peuvent être vrai ou faux. Par exemple `devantMur()` retourne vrai (`True` en Python) ou faux (`False`). Ces tests sont des procédures particulières qui calculent un résultat, on les appelle des *fonctions*.

En mathématique on écrit  $x = f(y)$ ,  $y$  joue ici le rôle de paramètre vu dans le paragraphe précédent,  $f$  est une fonction qui calcule une valeur et cette valeur est ensuite affectée à la variable  $x$ . En programmation une fonction va donc calculer une valeur et retourner cette valeur pour qu'elle puisse être utilisée, par exemple en l'affectant à une variable. Si la valeur retournée par la fonction est `resultat` on écrira `return resultat`, c'est bien sûr la dernière instruction exécutée puisque la fonction a réalisé son but : calculer un résultat.

Il y a donc une différence entre procédure et fonction :

- Une procédure *réalise* une action, il n'y a pas d'instruction `return`
- Une fonction *calcule* un résultat et doit le renvoyer avec l'instruction `return resultat`.

Bien qu'on puisse réaliser des actions dans une fonction il vaut mieux l'éviter pour faciliter la compréhension du programme.

Revenons au main du paragraphe précédent, nous avons écrit `carre(5)` sans nous soucier de la taille de la grille, il nous faudrait un moyen de calculer cette taille pour que le robot ne rencontre pas un mur sur son passage. Nous allons définir la fonction `nbCases` qui ne prend aucun paramètre et qui calcule le nombre de cases en hauteur ou en largeur : le robot est contre un mur dans une orientation donnée, doit avancer jusqu'au mur en comptant le nombre de cases `nb` puis revenir à son point de départ et reprendre son orientation initiale. La fonction doit ensuite renvoyer le résultat c'est à dire `nb`.

```
def nbCases():
    nb = 1
    while not devantMur() :
        avancer()
        nb = nb + 1
    demiTour()
    while not devantMur() :
        avancer()
    demiTour()
    #on retourne la valeur calculée
    return nb
```

Si le robot se trouve dans le coin sud ouest orienté Nord on peut connaître la hauteur et la largeur de la grille :

```
def main():
    init()
    hauteur = nbCases()
    droite()
    largeur = nbCases()
```

### 1.4. Affectations multiples.

Ici nous appelons 2 fois la fonction `nbCases` pour déterminer la hauteur puis la largeur, est-il possible qu'une fonction retourne les 2 valeurs en même temps ? Dans d'autres langages c'est impossible (simplement), en Python il suffit de renvoyer l'ensemble des deux valeurs qui sera affecté à deux variables : si `calculeTaille` calcule les deux valeurs `h` et `l` il suffit de retourner (dans l'ordre) le résultat sous la forme `return h, l` ou bien `return (h, l)`. On peut donc écrire : `hauteur, largeur = calculeTaille()` `hauteur` contiendra le 1<sup>o</sup> résultat (c'est à dire `h`) et `largeur` le 2<sup>o</sup>. Reprenons l'exemple précédent :

```
def calculeTaille():
    h = nbCases()
    droite()
    l = nbCases()
    #on retourne les 2 valeurs calculées
    gauche()
    return h, l
def main():
    init()
    hauteur, largeur = calculeTaille()
```

De la même manière on peut écrire `a, b = 1, 2`. 1 est affecté à `a` et 2 est affecté à `b`.

### 1.5. Fonctions avec paramètres.

Les procédures les fonctions peuvent avoir des paramètres comme les procédures : écrivons une fonction `min` qui retourne le minimum de deux variables `a` et `b`. Ce pourrait être par exemple la largeur et la hauteur de l'exemple précédent.

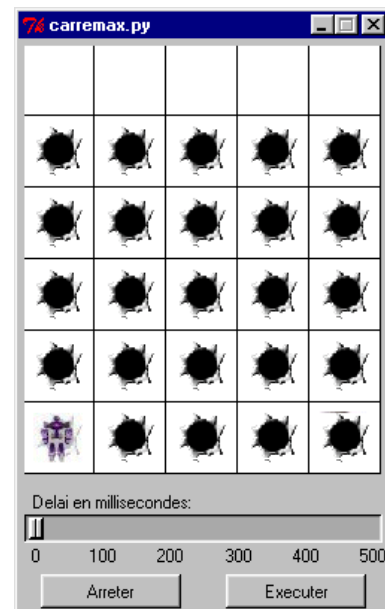
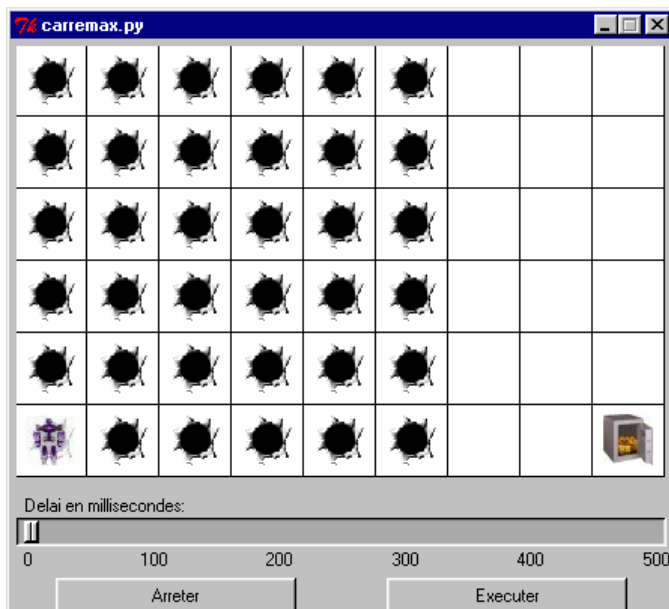
```
def min(a, b):
    if a < b :
        return a
    else :
        return b
```

Notons qu'ici nous avons 2 instructions `return` : à la 3<sup>e</sup> ligne la fonction est terminée et renvoie la valeur `a` si `a` est pas inférieur à `b` sinon les instructions suivantes sont exécutées et le résultat est égal à `b`.

### 1.6. Exercice de synthèse.

Illustrons à partir d'un exemple *carremax* tout ce que nous avons vu jusqu'à maintenant :

- Le robot est dans le coin sud ouest orienté Nord et doit décrire autant de carré que possible en partant du coin sud ouest (résultat ci-dessous).



```

def init(): #procédure sans paramètres
    initRobot(CSO, NORD)
def demiTour(): #procédure sans paramètres
    droite()
    droite()
def gauche(): #procédure sans paramètres
    droite()
    droite()
    droite()
def nbCases(): #fonction sans paramètres
    nb = 1
    while not devantMur() :
        avancer()
        nb = nb + 1
    #on revient au point de départ avec la même orientation
    demiTour()
    while not devantMur() :
        avancer()
    demiTour()
    #on retourne la valeur calculée
    return nb
def calculeTaille(): #fonction sans paramètres qui retourne 2 valeurs
    #on calcule la hauteur
    h = nbCases()
    droite()
    #on calcule la largeur
    l = nbCases()
    #orientation initiale
    gauche()
    #on retourne les 2 valeurs calculées
    return h, l
def min(a, b): #fonction avec paramètres
    if a < b :
        return a
    else :
        return b
def avancerDroite(x): #procédure avec paramètres
    nb = 0
    #avancer de x pas
    while nb < x :
        déposerMarque()
        avancer()
        nb = nb + 1
    droite()
def carre(k): #procédure avec paramètres
    nb = 0
    #4 côtés à parcourir
    while nb < 4 :
        avancerDroite(k-1)
        nb = nb + 1
def main(): #la procédure principale
    init()
    #affectation multiple
    hauteur, largeur = calculeTaille()
    nbCarres = min(hauteur, largeur)
    i = 1
    while i <= nbCarres :
        carre(i)
        i = i + 1

```

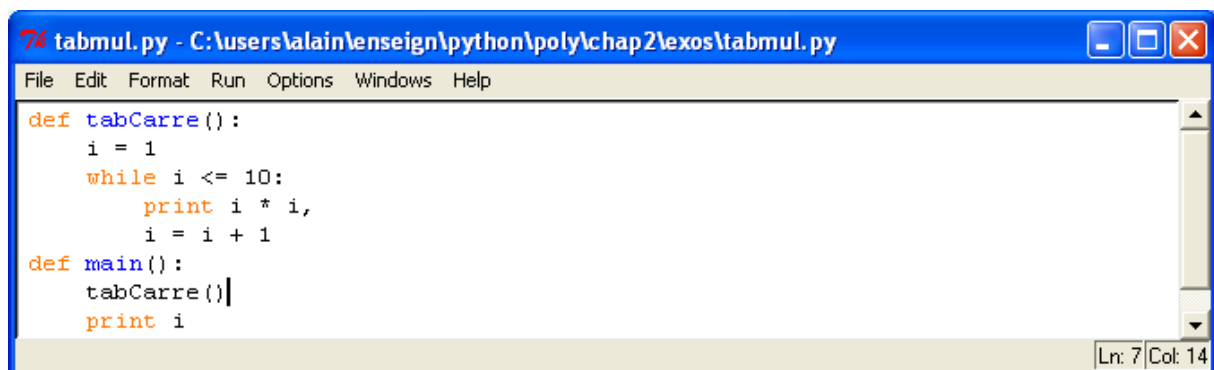
*carremax.py*

Dans cet exemple nous rencontrons :

- ✓ toutes les structures de contrôle :
  - la séquence.
  - la conditionnelle avec `if ... else` dans la fonction `min`.
  - la répétition avec le `while` dans la procédure `main`.
- ✓ des variables :
  - `nb` dans `carre`.
  - `hauteur`, `largeur` et `nbCarres` dans `calculeTaille()`.
- ✓ des fonctions et des procédures :
  - les procédures sans paramètre : `gauche()`.
  - les procédures avec paramètre : `carre(k)`.
  - les fonctions sans paramètres : `nbCases()` qui retourne une seule valeur.
  - les fonctions sans paramètres : `calculeTaille()` qui retourne deux valeurs.
  - les fonctions avec paramètres : `min(a, b)`.

## 2. Variables locales et globales.

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction. Chaque fois qu'on appelle une fonction python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel espace de noms. Le contenu des variables est stocké dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable `'i'` juste après avoir effectué la procédure `tabCarre`, nous obtenons un message d'erreur :

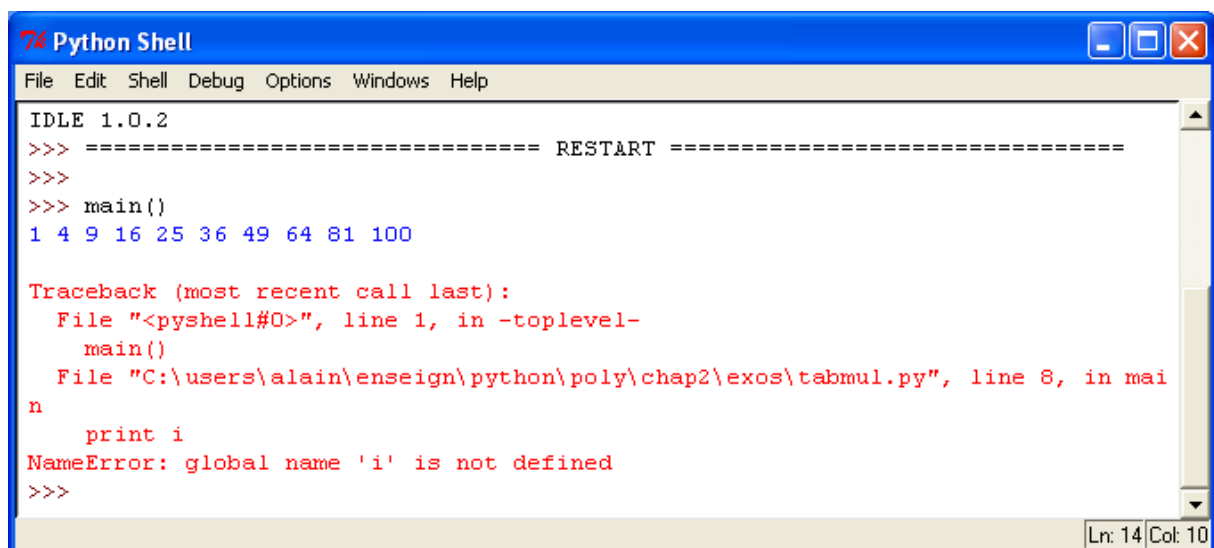


```

def tabCarre():
    i = 1
    while i <= 10:
        print i * i,
        i = i + 1
def main():
    tabCarre()
    print i
  
```

*tabmul.py*

Lançons l'interpréteur Python à partir d' Idle, exécutons `tabmul.py` puis appelons la méthode `main` :



```

IDLE 1.0.2
>>> ===== RESTART =====
>>>
>>> main()
1 4 9 16 25 36 49 64 81 100

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    main()
  File "C:\users\alain\enseign\python\poly\chap2\exos\tabmul.py", line 8, in mai
n
    print i
NameError: global name 'i' is not defined
>>>
  
```

*Idle*

Python nous signale que le symbole 'i' lui est inconnu, alors qu'il était correctement utilisé par la procédure `tabCarre`. L'espace de noms qui contient le symbole 'i' est strictement réservé au fonctionnement interne de `tabCarre`, et il est automatiquement détruit dès que la procédure a terminé son travail. La variable 'i' est dite locale à `tabCarre`.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est "visible" de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier. Exemple :

```
1 a, b = 2, 3
2 def testLocalGobal():
3     a = 5
4     print a, b
5 def main():
6     print a, b
7     testLocalGobal()
8     print a, b
>>> main()
2 3
5 3
2 3
```

Dans cet exemple nous commençons par définir à la ligne 1 deux variables globales `a` et `b`. Puis nous définissons une procédure `testLocalGobal` qui définit la variable 'a' avec 5 comme valeur initiale. Nous avons donc défini 2 variables différentes : une interne à la procédure qui vaut 5 et une autre globale qui vaut 2. Lors de l'affectation 'a = 5' le contenu de la variable globale n'est pas modifié mais une variable locale est créée. Il y a donc deux variables 'a'. A la ligne 4 la procédure affiche le contenu de la variable locale 'a' et de la variable globale 'b'. Dans le `main` à la ligne 6 on affiche le contenu des variables globales 'a' et 'b', en les réaffichant à la ligne 8 après l'appel de la procédure `testLocalGlobal` on vérifie que le contenu de la variable globale 'a' est inchangé.

Comment alors modifier une variable globale puisqu'une affectation dans une fonction crée une nouvelle variable locale à la fonction ? Il suffit d'indiquer à Python de ne pas créer de variable locale mais d'utiliser la variable globale à l'aide de l'instruction `global`:

```
a, b = 2, 3
def testLocalGobal():
    global a
    a = 5
    print a, b
def main():
    print a, b
    testLocalGobal()
    print a, b
>>> main()
2 3
5 3
5 3
```

Cette fois-ci il n'y a qu'une seule variable 'a' qui est globale et la procédure `testLocalGobal` a changé son contenu.

### 3. Modules de fonctions

Il est intéressant de regrouper des fonctions apparentées sous forme de modules. Ces modules constituent des bibliothèques réutilisables : les fonctions sont définies dans les modules pour être utilisées dans un programme. Définissons par exemple un module `actionsElementaires.py` :

```
def demiTour():
    droite()
    droite()
def gauche():
    droite()
    droite()
    droite()
def jusquAuMur():
    while not devantMur():
        avancer()
```

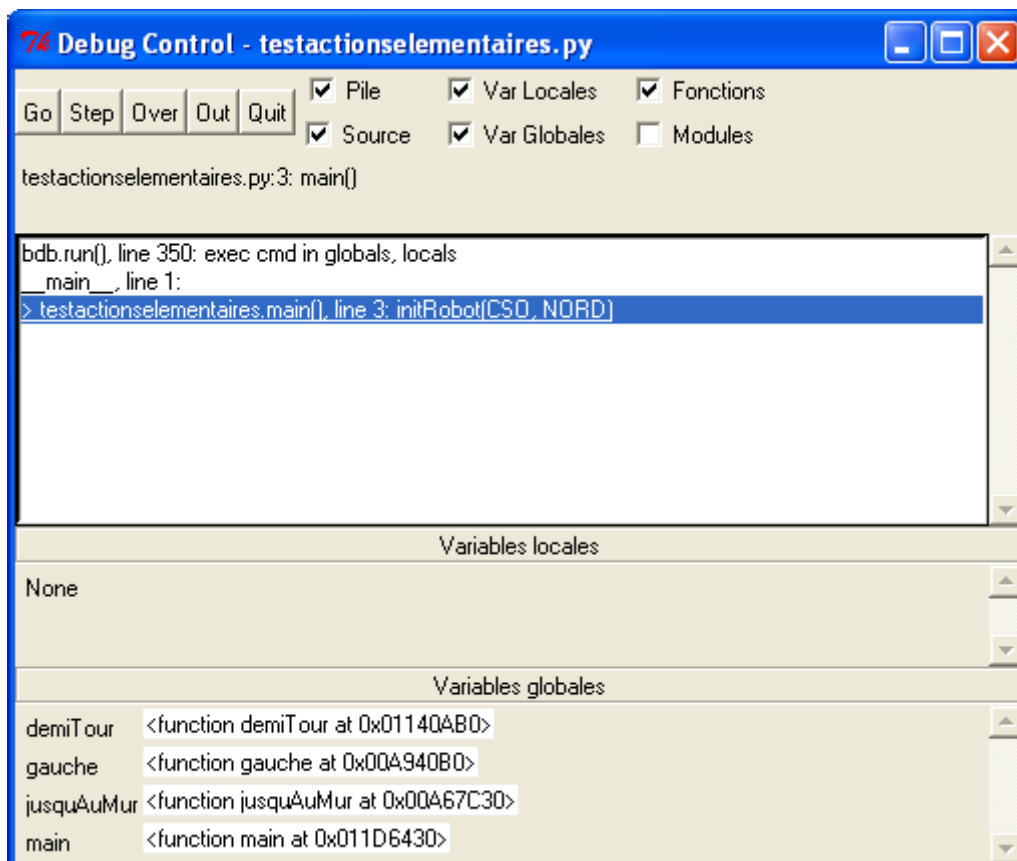
*actionsElementaires.py*

On peut maintenant utiliser ce module de deux manières :

- en important toutes les fonctions définies dans le module au moyen de l'instruction

`from module import *` Les fonctions sont alors directement utilisables : on pourra écrire simplement `demiTour()`

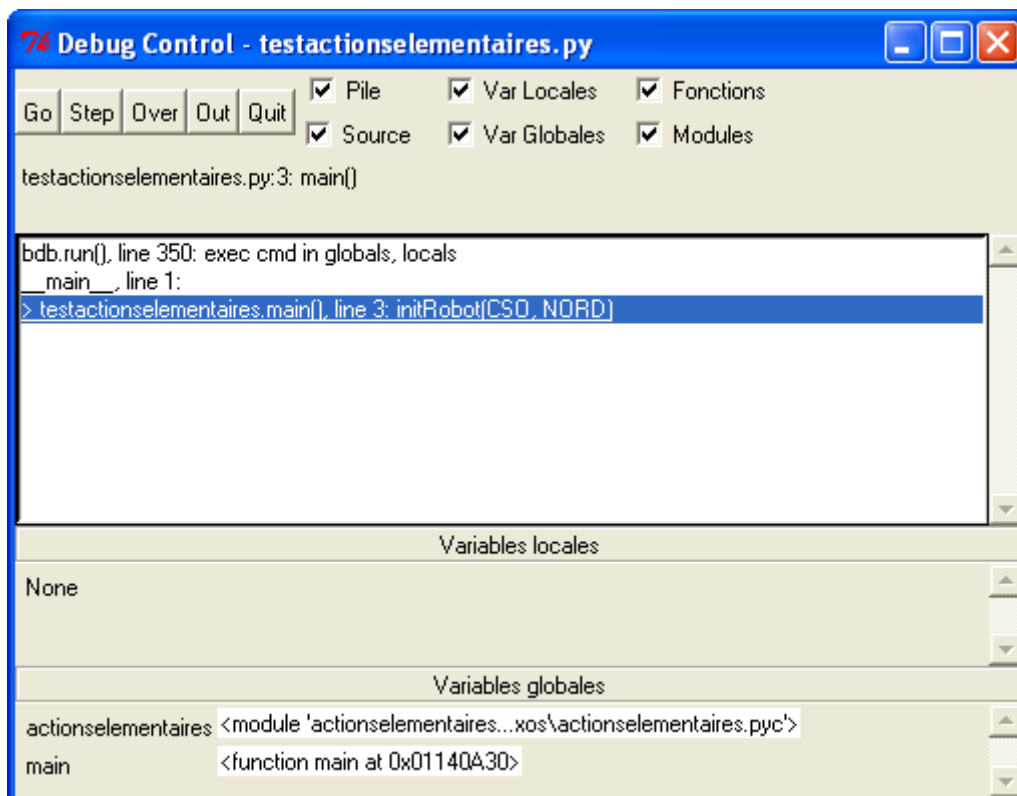
```
from actionsElementaires import *
def main():
    initRobot(CSO, NORD)
    jusquAuMur()
    demiTour()
    jusquAuMur()
    demiTour()
```



On voit ici que les procédures `jusquAuMur` et `demiTour` ont été importée et que les variables contiennent les adresses du code exécutable correspondant.

- en important le module lui même au moyen de l'instruction `import module`. Il faudra alors indiquer à Python que la fonction que l'on désire utiliser se trouve dans ce module, l'appel de la fonction s'écrira `actionsElementaires.demiTour()`

```
import actionsElementaires
def main():
    initRobot(CSO, NORD)
    actionsElementaires.jusquAuMur()
    actionsElementaires.demiTour()
    actionsElementaires.jusquAuMur()
    actionsElementaires.demiTour()
```



En dehors de la manière d'appeler les fonctions il existe une différence entre ces deux méthodes, préférez la seconde : il vaut mieux importer le module plutôt que les fonctions.



### 1.3. Fonctions intégrées et modules.

- Python offre des fonctions intégrées :

<code>int(x)</code>	<code>int(2.5) -&gt; 2</code>
<code>long(x)</code>	<code>long(2.5) -&gt; 2L</code>
<code>float(x)</code>	<code>float(2) -&gt; 2.0</code>
<code>min(i, ....)</code>	<code>min(5,2,6,9,7) -&gt; 2</code>
<code>max(i, ...)</code>	<code>max(5,2,6,9,7) -&gt; 9</code>
<code>abs(x)</code>	<code>abs(-2.5) -&gt; 2.5</code>

Il va de soi qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des modules que l'on a vu au chapitre précédent. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées que l'on appelle des *librairies*.

- Le module `math` contient également des constantes et fonctions utiles. Pour l'utiliser il faut l'importer avec l'instruction `import` :

```
>>> import math
>>> print math.pi
3.1415926535897931
>>> print math.sqrt(2)
1.4142135623730951
```

## 2. Les booléens.

### 2.1. Le type bool.

Ce type est constitué de 2 valeurs : `True` et `False`. Ce sont les valeurs renvoyées par les tests du robot devant `Marque()` ou devant `Mur()`. Les opérateurs de comparaisons `'>'` `'=='` retournent des valeurs de ce type:

```
>>> print 1>2
False
```

### 2.2. Opérateurs.

Ils ont déjà été abordé lors du précédent chapitre, rappelons les:

- `not x` qui inverse la valeur de `x` :
- `x and y` qui prend la valeur `True` si `x` et `y` sont tous les deux égaux à `True` :
- `x or y` qui prend la valeur `True` si une des valeurs `x` ou `y` est égale à `True` :

```
>>> print not(1 > 2)
True
>>> print (2>1) and (5>4)
True
>>> print (1>2) and (5>4)
True
```

### 2.3. Exercice.

Les caractères se notent 'a' ou 'b', écrire un programme qui demande à l'utilisateur d'entrer un caractère et qui retourne vrai si ce caractère est 'o' ou 'O' et faux sinon. On utilisera la fonction `input("message")` qui retourne le caractère lu.

Version 1 :

```
def lire() :
    s = input("Entrez une lettre: ")
    if s == 'o' or s == 'O':
        return True
    else:
        return False
>>> lire()
Entrez une lettre: 'o'
True
>>>
```

Ce programme est juste mais on remarque que l'on savait déjà la réponse au moment du test : ce programme est équivalent à écrire :

```
Si test == vrai
Alors return vrai
Sinon return false
```

On peut évidemment écrire de manière plus concise : `return test` d'où cette deuxième version :

```
def lire() :
    s = input("Entrez une lettre: ")
    return s == 'o' or s == 'O'
```

Ce programme souffre encore d'une imperfection : l'utilisateur doit entrer 'o' il serait préférable d'entrer uniquement la lettre, la fonction `raw_input("message")` lit l'entrée et effectue la conversion vers le type approprié (mais cela ne convient pas toujours).

Version 3 :

```
def lire() :
    s = raw_input("Entrez une lettre: ")
    return s == 'o' or s == 'O'
>>> lire()
Entrez une lettre: o
True
>>>
```

## 3. Chaînes de caractères.

### 3.1. Le type *str*.

Une donnée de type `str` est une suite quelconque de caractères délimitée soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). Le type `char` n'existe pas en Python.

```
>>> s1 = 'un'
>>> s2 = '"deux" trois'
>>> s3 = "quatre"
>>> print s1, s2, s3
'un "deux" trois quatre'
>>> type(s1)
<type 'str'>
```

On peut délimiter la chaîne à l'aide de triples guillemets ou de triples apostrophes `"""` pour écrire des chaînes de caractères sur plusieurs lignes.

Certains caractères comme les tabulations sont codés suivant une séquence (entre apostrophes):

Caractère	Séquence
nouvelle ligne	\n
tabulation	\t
saut de page	\f
backslash	\\
apostrophe	\'
guillemet	\"
retour en arrière	\b
retour chariot	\r

### 3.2. Opérateurs.

- Concaténation :

On peut assembler plusieurs petites chaînes pour en construire de plus grandes. Cette opération s'appelle concaténation et on la réalise à l'aide de l'opérateur '+' (Cet opérateur réalise donc l'opération d'addition lorsqu'on l'applique à des nombres, et l'opération de concaténation lorsqu'on l'applique à des chaînes de caractères :

```
>>> s1 = 'un'
>>> s2 = 'deux'
>>> s3 = s1 + s2
>>> print s3
'un deux'
```

- Répétitions :

On peut également répéter une chaîne de caractères avec l'opérateur '\*':

```
>>> s1 = 'un'
>>> s2 = s1*3
>>> print s2
'ununun'
```

- Indiçage :

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des *collections*. Une collection est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. En fonction des circonstances, nous souhaiterons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères distincts

Python considère qu'une chaîne de caractères est un objet de la catégorie des *séquences*, lesquelles sont des collections *ordonnées* d'éléments. Les caractères de la chaîne sont ainsi toujours disposés dans un certain ordre. Chaque caractère d'une chaîne peut donc être désigné par sa place dans la séquence, à l'aide d'un index, pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne, et on lui accole entre deux crochets l'index numérique qui correspond à la position du caractère dans la chaîne : `s[2]`

**Attention**, les chaînes de caractères sont numérotées *à partir de zéro* (et non à partir de un).

```
>>> ch = "Le Robot"
>>> print ch[0], ch[3]
L R
```

On ne peut pas modifier une chaîne de caractères, on ne peut donc pas écrire `ch[0] = 'a'`

- Exercice : Demander à l'utilisateur d'entrer un mot puis l'afficher à l'envers

```
def palindrome() :
    s = raw_input("entrez un mot: ")
    i = len(s) - 1
    while i >= 0:
        print s[i],
        i = i - 1
    print
>>> palindrome()
entrez un mot: abc
c b a
```

On affiche lettre par lettre sans retourner à la ligne avec l'expression `print x`, l'inconvénient de ce programme est bien sûr que les lettres sont espacées. Ecrivons un 2<sup>o</sup> programme qui construit d'abord le mot à l'envers puis l'affiche :

```
def palindrome() :
    s = raw_input("entrez un mot: ")
    inv = ""
    i = len(s) - 1
    while i >= 0:
        inv = inv + s[i]
        i = i - 1
    print inv
>>> palindrome()
entrez un mot: abc
cba
>>>
```

### 3.3. Fonctions intégrées.

La fonction intégrée `len` renvoie la longueur d'une chaîne, `str(x)` permet de convertir un numérique en chaîne. Les fonctions de conversion `int`, `float` permettent l'opération inverse.

```
>>> print len('abcde')
5
>>> print str(5)
'5'
>>> print int('5')
5
>>> print '5' + 5
TypeError: cannot concatenate 'str' and 'int' objects
>>> print int('5') + 5
10
>>> print '5' + str(5)
'55'
>>>
```

### 3.4. Exercice : voyelles

- Demander à l'utilisateur d'entrer un mot puis afficher le nombre de voyelles

On peut écrire cette première version en utilisant `if/elif` vue en TP :

```

def compter():
    s = raw_input("entrez un mot: ")
    nb = 0
    i = 0
    while i < len(s):
        if s[i] == 'a':
            nb = nb + 1
        elif s[i] == 'e':
            nb = nb + 1
        elif s[i] == 'i':
            nb = nb + 1
        elif s[i] == 'o':
            nb = nb + 1
        elif s[i] == 'u':
            nb = nb + 1
        elif s[i] == 'y':
            nb = nb + 1
        i = i + 1
    print nb
>>> compter ()
entrez un mot: homer
2

```

Cette première version n'est pas convainquante, que faire pour les consonnes ? Essayons en utilisant une double boucle while :

```

def compter():
    s = raw_input("entrez un mot: ")
    voyelles = "aeiouy"
    nb = 0
    i = 0
    while i < len(s):
        j = 0
        while j < len(voyelles):
            if s[i] == voyelles[j]:
                nb = nb + 1
            j = j + 1
        i = i + 1
    print nb

```

Cette seconde version n'est pas plus convainquante que la première, il vaudrait mieux dire : « si la lettre d'indice i est contenue dans l'ensemble des voyelles alors incrémenter nb »

### 3.5. Test d'appartenance d'un élément à une séquence.

On peut tester si une lettre 'c' appartient à une chaîne de caractère str avec l'instruction 'c in str' qui retourne True si elle est présente et False sinon. D'où cette troisième version :

```

def compter():
    s = raw_input("entrez un mot: ")
    voyelles = "aeiouy"
    nb = 0
    i = 0
    while i < len(s):
        if s[i] in voyelles:
            nb = nb + 1
        i = i + 1
    print nb

```

Cette fois c'est un peu mieux mais on aimerait bien pouvoir écrire :

```

Pour chaque lettre du mot s :
    Si cette lettre appartient à l' ensemble des voyelles :
        Alors incrémenter nb

```

### 3.6. Parcours d'une séquence.

Le parcours d'une séquence est une opération tellement fréquente en programmation que Python propose une structure de boucle plus appropriée, basée sur le couple d'instructions **for ... in ...** :

On peut écrire :

```

for c in str:
    print c ,

```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle évite d'avoir à définir et à incrémenter une variable spécifique pour gérer l'indice du caractère que l'on veut traiter à chaque itération. La variable `c` contiendra successivement tous les caractères de la chaîne, du premier jusqu'au dernier.

On peut donc réécrire le programme précédent de manière plus naturelle :

```

def compter():
    s = raw_input("entrez un mot: ")
    nb = 0
    voyelles = "aeiouy"
    for c in s:
        if c in voyelles:
            nb = nb + 1
    print nb

```

### 3.7. Modules et méthodes.

Le module `string` fournit des outils intéressants comme `upper` ou `lower`:

```

>>> import string
>>> s = 'Un'
>>> print string.upper(s)
'UN'
>>> print string.lower(s)
'un'
>>> print s
'Un'

```

Les modules constituent des bibliothèques de constantes ou de fonctions que l'on applique à une variable, nous avons vu par exemple `math.sqrt(5)` ou `string.lower('Un')`. Dans le cas des objets comme les chaînes de caractères nous pouvons demander à un objet précis d'exécuter une fonction qu'il connaît. Nous verrons en détail les objets dans un chapitre à part mais nous pouvons déjà expliquer le principe : Prenons des voitures, il est clair qu'elles ont toutes des propriétés différentes comme la couleur, la marque, l'immatriculation etc mais elles savent toutes comment démarrer, avancer, reculer etc ... , ces actions démarrer etc sont les *méthodes* que nous pouvons demander à une voiture d'exécuter. Soit une voiture `v1`, pour lui demander d'avancer on écrira `v1.avancer()`. Les chaînes de caractères possèdent également des méthodes :

- `s.upper()` retourne une copie de la chaîne de caractère `s` en majuscule
- `s.lower()` retourne une copie de `s` en minuscule
- `s.find(c)` retourne l'index de `c` dans `s` ou `-1` si `c` n'est pas présent
- `s.index(c)` retourne l'index de `c` dans `s` ou provoque une erreur si `c` n'est pas présent

```

>>> print s.upper()
'UN'
>>> print s.lower()
'un'
>>> print s.find('n')
1
>>> print s.find('z')
-1

```

### 3.8. Les tranches

Supposons que nous demandions à un utilisateur d'entrer son nom et prénom et que nous voulions les afficher avec la première lettre en majuscule. La méthode `upper` nous permet de convertir la première lettre mais il faudrait ensuite une boucle pour écrire les autres lettres. Les tranches permettent d'extraire une partie de la liste : On indique entre crochets les indices correspondant au début (inclus) et à la fin (exclus) de la "tranche" que l'on souhaite extraire :

```
>>> str = "Homer"
>>> print str[1:4]
ome
>>> print str[:4]
Home
>>> print str[1:]
omer
>>> print str[:]
Homer
```

On peut donc maintenant écrire très facilement notre programme:

```
def nomPrenom():
    nom,prenom = input("entrez le nom et le prénom ('Nom','prenom'):")
    s = nom[0].upper() + nom[1:] + " " + \
        prenom[0].upper() + prenom[1:]
    print s
>>> nomPrenom()
entrez le nom et le prénom ('Nom','prenom') : ("homer","pizza")
Homer Pizza
```

## 4. Listes.

### 4.1. Le type list.

Une liste est une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets.

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800]
>>> print jour
['lundi', 'mardi', 'mercredi', 1800]
>>> print type(jour)
<type 'list'>
```

Dans cet exemple, la valeur de la variable `jour` est une liste. La liste vide est représentée par `[]`.

Comme on peut le constater dans l'exemple choisi, les éléments qui constituent une liste peuvent être de types variés. Dans cet exemple, en effet, les trois premiers éléments sont des chaînes de caractères, le quatrième élément est un entier, le cinquième un réel, etc. (Nous verrons plus loin qu'un élément d'une liste peut lui-même être une liste !). Le concept de liste est donc assez différent du concept de "tableau" que l'on rencontre dans d'autres langages de programmation.

On peut obtenir une liste constituée des caractères d'une chaîne de caractères `s` à partir de l'instruction `list(s)` :

```
>>> print list("lundi")
['l', 'u', 'n', 'd', 'i']
```

### 4.2. Opérateurs.

Comme pour les chaînes de caractères les opérateurs concaténation `+` et répétition `*` et indiciage `[]` sont applicables :

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l3 = l1 + l2
>>> print l3
```

```
[1, 2, 3, 4, 5, 6]
>>> print l1*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print l1[1]
2
```

A la différence de ce qui se passe pour les chaînes, qui constituent un type de données non-modifiables il est possible de changer les éléments individuels d'une liste :

```
>>> print jour
['lundi', 'mardi', 'mercredi', 1800]
>>> jour[3] = jour[3] + 47
>>> print jour
['lundi', 'mardi', 'mercredi', 1847]
```

On peut donc remplacer certains éléments d'une liste par d'autres :

```
>>> jour[3] = 'jeudi'
>>> print jour
['lundi', 'mardi', 'mercredi', 'jeudi']
```

### 4.3. Fonction intégrées.

La fonction intégrée `len()`, que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes, elle renvoie le nombre d'éléments présents dans la liste :

```
>>> print len(jour)
7
```

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

```
>>> del(jour[3])
>>> print jour
['lundi', 'mardi', 'mercredi']
```

### 4.4. Méthodes de listes

Les listes sont des objets comme les chaînes de caractères, elles possèdent donc des méthodes :

- `l.append(obj)` ajoute l'objet `obj` en fin de la liste `l`.
- `l.remove(obj)` enlève l'objet `obj` ayant l'indice le plus petit de la liste `l` ou provoque une erreur s'il est pas présent.
- `l.index(obj)` retourne l'index de `obj` dans `l` ou provoque une erreur si `obj` n'est pas présent.
- `l.count(obj)` retourne le nombre de fois `obj` est présent dans `l` ou 0 si `obj` n'est pas présent.
- `l.reverse()` inverse la liste `l`
- `l.sort()` trie la liste `l` dans l'ordre 'naturel »

```
>>> jour.append('jeudi')
>>> jour.remove('mardi')
>>> print jour
['lundi', 'mercredi', 'jeudi']
>>> print jour.index('jeudi')
2
>>> print jour.count('jeudi')
1
>>> jour.reverse()
>>> print jour
['jeudi', 'mercredi', 'lundi']
>>> jour.sort()
>>> print jour
['jeudi', 'lundi', 'mercredi']
```

#### 4.5. Exercice 1 : la méthode range

Ecrire la fonction `tableMultiplication(x)` qui construit la liste constitué par la table de multiplication de `x` (les 10 premiers multiples).

Nous pouvons commencer en écrivant une boucle `while` :

```
def tableMultiplication(x):
    lst = []
    n = 0
    while n <= 10:
        lst.append( n * x)
        n = n + 1
    return lst
>>> print tableMultiplication(8)
[0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80]
>>> print tableMultiplication(8)[2]
16
```

Cette manière de faire impose de gérer soi-même l'incréméntation de `n`, rappelons nous du parcours d'une séquence, nous avons écrit :

```
for c in str:
    print c
```

Dans le même ordre d'idée nous pourrions écrire :

```
def tableMultiplication(x):
    lst = []
    for n in [0,1,2,3,4,5,6,7,8,9,10]:
        lst.append( n * x)
    return lst
```

Ceci nous oblige toutefois à écrire la liste `[0,...,10]` ce qui est un peu fastidieux, heureusement Python met à notre disposition la fonction `range(debut, fin)` qui retourne la liste constituée des entiers successifs de `debut` (inclus) à `fin` (exclus) :

```
>>> print range(0,11)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Nous pouvons donc réécrire la fonction `tableMultiplication(x)` d'une manière plus « pythonienne » :

```
def tableMultiplication(x):
    lst = []
    for n in range(0,11):
        lst.append( n * x)
    return lst
```

#### 4.6. Exercice 2 : le module random

Lorsque nous avons vu les données numériques nous avons introduit le module `math` mais nous n'avons pas pu parlé du module `random` car il faisait intervenir les listes. La méthode `choice(uneListe)` du module `random` retourne un élément de la liste `uneListe` au hasard :

```
>>> print jour
['jeudi', 'lundi', 'mercredi']
>>> print random.choice(jour)
'mercredi'
>>> print random.choice(range(0,11))
6
```

Ecrire la fonction `sommeHasard()` qui affiche une liste triée de 4 entiers choisis au hasard entre 0 et 10 puis qui affiche leur somme.

```

import random
def sommeHasard():
    l = []
    for i in range(0,4):
        l.append(random.choice(range(0,11)))
    l.sort()
    print l
    somme = 0
    for i in l:
        somme = somme + i
    print somme
>>> sommeHasard()
[0, 7, 8, 9]
24

```

## 5. Exercice de synthèse : le jeu du pendu.

Vous connaissez tous le jeu du pendu : il faut deviner un mot en proposant les lettres une par une, si le joueur n'a pas trouvé le mot au bout de 9 essais il a perdu.

### 5.1. Première version.

The screenshot shows a Python IDE window titled 'pendu.py' with a menu bar (File, Edit, Format, Robot, Configuration, Windows, Help). The code in the editor includes a function 'jouer()' that manages the game logic, including choosing a word, displaying it with dashes, and checking for correct letters. A 'Debug Control' window is overlaid on the code, showing the execution stack and local variables. The stack trace indicates the current execution point is at 'pendu.jouer(), line 36: c = raw\_input("entrez une lettre: ")'. The local variables table shows: c is 'o', i is 0, nbVies is 8, pasTrouve is ['-', 'e', 'n', 'd', 'u'], rep is 'pendu', and trouve is ['p', '-', '-', '-', '-']. Below the IDE, a terminal window shows the game's output: '-----', 'entrez une lettre: p', '\*\*\*\*\*', 'p-----', 'entrez une lettre: o', '\*\*\*\*\*', 'p-----'.

Le principe est de travailler sur deux listes :

- la liste des lettres trouvées `trouve`, au départ le joueur n' rien trouvé donc on placera ls caractère '-'.
- celle des lettres restant à trouver `pasTrouve` qui contient au départ toutes les lettres du mot à trouver.

Nous travaillons sur des listes et non sur des chaînes de caractères car elles ne sont pas modifiables. Supposons que le mot à trouver soit "pendu" et que l'utilisateur ait proposé les lettres 'p' et 'o'

- le mot à trouver sera mémorisé dans la variable `rep` qui contient donc 'pendu' (c'est une chaîne de caractère car on ne la modifiera pas)
- la variable `trouve` contiendra ['p', '-', '-', '-', '-']
- la variable `pasTrouve` contiendra ['-', 'e', 'n', 'd', 'u']

Pour déterminer si le joueur a gagné il suffit de vérifier si la variable `pasTrouve` ne contient que le caractère '-'

La capture d'écran montre l'état des variables ainsi que l'affichage dans le terminal..

```
import random
def choisirMot():
    mots = ("pendu", "vivant")
    return random.choice(mots)
def afficheMot(listeCar):
    s = ""
    for i in listeCar:
        s = s + i
    print s
def afficheNbVies(nb):
    print '*' * nb
def gagne(listeCar):
    return listeCar.count('-') == len(listeCar)
def jouer():
    rep = choisirMot()
    pasTrouve = list(rep)
    trouve = ['-'] * len(rep)
    nbVies = 9
    fini = False
    while nbVies > 0 and not fini:
        afficheNbVies(nbVies)
        afficheMot(trouve)
        c = raw_input("entrez une lettre: ")
        if c in pasTrouve:
            i = pasTrouve.index(c)
            pasTrouve[i] = '-'
            trouve[i] = c
            fini = gagne(pasTrouve)
        else:
            nbVies = nbVies - 1
    if fini:
        print rep, "felicitations"
    else:
        print "perdu, le mot etait:", rep
def continuer():
    c = raw_input("Voulez vous continuer O/N: ")
    return c == 'o' or c == 'O'
def main():
    jouer()
    while continuer():
        jouer()
```

*Pendul.py*

## 5.2. Interruption d'une boucle : break

Dans la procédure `jouer()` nous avons introduit un booléen pour pouvoir sortir de la boucle `while` lorsque l'utilisateur avait trouvé le mot. Il est possible de sortir d'une boucle (`while` ou `for`) à tout instant grâce à l'instruction `break` :

```
while <condition 1> :
    instructions
    if <condition 2> :
        break
    instructions
    if <condition 3>:
        break
    etc.
```

Si les condition 1 ou 2 sont vérifiées alors on sort de la boucle pour passer à la suite du programme. Utilisons le `break` pour réécrire la procédure `jouer()` : la variable `fini` n'est donc plus utile, pour déterminer en sortie de boucle si le joueur a gagné lors de l'affichage du résultat il faudra maintenant utiliser la fonction `gagne()`

```
def jouer():
    rep = choisirMot()
    pasTrouve = list(rep)
    trouve = ['-'] * len(rep)
    nbVies = 9
    while nbVies > 0:
        afficheNbVies(nbVies)
        afficheMot(trouve)
        c = raw_input("entrez une lettre: ")
        if c in pasTrouve:
            i = pasTrouve.index(c)
            pasTrouve[i] = '-'
            trouve[i] = c
            if gagne(pasTrouve):
                break #on sort de la boucle while
        else:
            nbVies = nbVies - 1
    if gagne(pasTrouve):
        print rep, "felicitations"
    else:
        print "perdu, le mot etait:", rep
```

*Pendu2.py*