# 1 Notion de langage de programmation

**Définition 1**

Un procédé systématique qui permet de traiter des informations sous la forme d'une série d'instructions à exécuter s'appelle **un algorithme**. Ce mot vient du nom du mathématicien Al-Khwarizmi.
Une recette de cuisine, le calcul de la longueur de l'hypoténuse d'un triangle rectangle, une technique de calcul multiplication, sont des algorithmes.
Au XX$^{ème}$ siècle, on a inventé des machines capables d'exécuter des algorithmes, ce sont les ordinateurs. Un programme est un texte qui décrit un algorithme que l'on veut faire exécuter par un ordinateur. L'ordinateur ne comprend que le langage machine suite binaire de 0 et de 1, difficilement manipulables par l'homme, même si les **langages d'assemblage** en donnent des représentations compréhensibles.
En pratique, on écrit le code source du programme dans un **langage de programmation de haut niveau** (C, Python, Java ...) sous la forme d'un fichier texte puis un **compilateur** ou un **interpréteur** le transforme en une suite d'instructions compréhensibles par la machine.

- dans les **langages compilés** (C), le code source doit être compilé en un fichier binaire par un compilateur qui dépend de l'architecture de la machine puis on exécute ce fichier binaire ;
- dans les **langages interprétés** (Perl), un interpréteur transcrit les instructions du code source en langage machine à la volée, au fur et à mesure de son exécution ;
- dans les **langages semi-interprétés**, le fichier source est transformé en un code intermédiaire le `byte-code` qui est ensuite interprété. En Java les deux étapes sont distinctes, le `byte-code` étant exécuté par une machine virtuelle. En Python, les deux opérations sont effectuées à la volée par l'interpréteur : le texte du programme est d'abord traduit en `byte-code`, puis ce dernier est exécuté. On peut le constater si `programme1.py` est importé dans un autre programme `programme2.py`. Python crée lors de la première exécution de `programme2.py`, un répertoire `__pycache__` contenant un fichier `programme1.cpython-32.pyc` contenant le `byte-code` de `programme1.py`. Une partie du travail sera donc faite pour une prochaine exécution de `programme2.py`.

Les langages compilés sont plus rapides et les langages interprétés ont une meilleure portabilité sur les différentes architectures de machines. La plupart des langages de programmation comportent des constructions fondamentales : l**a déclaration et l'affectation de variables, la séquence, le test, la boucle, la fonction** ...C'est le **noyau impératif** du langage. Il existe d'autres styles (ou paradigmes) de programmations (objets, fonctionnelle ...).

**Remarque 1**

***Python*** *est un langage semi-interprété multi-paradigme (impératif, objets ...) créé par Guido Van Rossum au début des années 90. Il est sous licence libre GPL et gratuit et fonctionne sur toutes les plateformes (Windows, Linux, OSX ...). Actuellement les versions courantes sont Python 2.7.3 et Python 3.2.3, l'évolution du langage étant assurée par la Python Software Foundation à but non lucratif. Attention la branche 3.x de Python n'est pas totalement rétrocompatible avec la branche 2.x.*
*Quelques liens vers des ressources :*

- *Site officiel :* `http://www.python.org/` `http://www.python.org/download/` `http://www.python.org/doc/`
- *Cours sur Python :* `http://inforef.be/swi/python.htm` *ou* `http://python.developpez.com/cours/`
- *Tutoriel sur Python :* `http://www.siteduzero.com/tutoriel-3-223267-apprenez-a-programmer-en-python.html`

*En Python on peut tester des instructions en mode console mais on composera les programmes (ou scritps) plus longs avec un éditeur de texte (éditeur d'Idle, Notepad+ sous Windows, Geany ...).*
*Nous travaillerons avec Python 2.7 et et Python 3.2 car certaines bibliothèques (comme* `Pygame`*) n'ont pas été encore portées sur la branche 3.x.*
*Python est installé par défaut sur la distribution Linux Ubuntu, sous Windows on peut télécharger les installateurs à partir de* `http://www.python.org/download/` *et on disposera d'un environnement graphique Idle avec console et éditeur de texte.*

# 2 La console Python

On peut utiliser la console Python (celle fournie par `Idle`, ou avec la commande `python` ou `python3` dans une console si Python est installé ), comme une calculatrice. On saisit une `expression` ou une `instruction` après les trois >>> de l'invite de commande, on appuie sur Entrée et Python l'interprète. Si c'est une expression il applique la fonction `eval()` pour l'évaluer.

Une **expression est une suite de caractères qui peut définir une valeur pour l'interpréteur**. Elle peut être constituée d'opérations sur des constantes (nombres, booleens,chaines de caratcères) ou des variables (noms associés à des valeurs variables).
On peut évaluer plusieurs expressions sur une même ligne en les séparant par une « , ».

Exécuter le code suivant :

```
>>> print('Hello World')
Hello World
>>> 3*5+7-(10+3)
9
>>> 2**3 # pour l'exponentiation (et non pas 2^3)
8
>>> 2,5/3, 2.5/3
(2, 1.6666666666666667)
0.8333333333333334
>>> 5/2, 5//2, 5%2 #Division exacte, Quotient et reste de la division euclidienne
(2.5, 2, 1)
```

Tout ce qui se situe à droite d'un symbole « dièse » # jusqu'au changement de ligne, est ignoré par l'interpréteur. On peut ainsi insérer des **commentaires**.

# 3 Expressions et variables

Exécuter les instructions suivantes :

```
>>> a = 12 #On affecte à la variable nommée a la valeur 12, le symbole d'affectation est =
>>> a, id(a), type(a) #Valeur de a, Identifiant mémoire de a et Type de a
(12, 137396176, <class 'int'>)
>>> b, c = a, 2*a #On peut affecter en parallèle la valeur de a à la variable b et la valeur
      2*a à la variable c
>>> b, id(b), type(b) # b = a donc b a même valeur et même identifiant que a
(12, 137396176, <class 'int'>)
>>> c, id(c), type(c)
(24, 137396368, <class 'int'>)
>>> a = 0
>>> a, id(a), type(a) #Lorsqu'on change la valeur de a, on change son identifiant
(0, 137395984, <class 'int'>)
>>> b, id(b), type(b) #En revanche la valeur et l'identifiant de b ne changent pas tant que
     b ne subit pas une nouvelle affectation
(12, 137396176, <class 'int'>)
>>> m=10;n=100;p=1000;q=10000;r=100000 #le ; est le séparateur d'instruction
>>> m=m+1;n+=1;p-=1;q*=2;r/=2 #Pour incrémenter une variable on peut utiliser le raccourci n
     +=1 à la place de n = n+1. Des raccourcis similaires existent avec les opérateurs -,*
     et /
>>> m,n,p,q,r
(11, 101, 999, 20000, 50000.0)
>>> del a #La primitive del permet de supprimer une variable
```

**Définition 2**

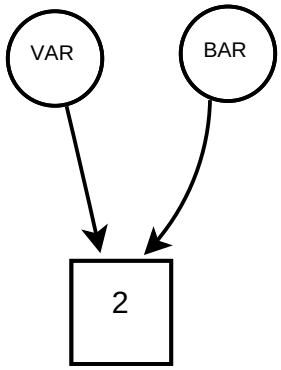
Dans un programme Python, comme dans d'autres langages, l'ensemble des données en mémoire à un instant $t$ s'appelle **l'état courant du programme**.
Le programme utilise des **constantes** mais il peut aussi modifier certaines données variables. Une **variable** est l'association entre un **nom de variable** et une case mémoire contenant **la valeur de la variable**.
Pour créer une variable, on doit d'abord faire une **déclaration de variable** qui établit un lien entre le nom de la variable et une case mémoire qui contiendra sa valeur lorsque la variable sera affectée.
La valeur d'une variable est modifiée par une **affectation de variable**. En Python, pour affecter à la variable de nom `var` la valeur entière 2, on écrit `var = 2` .`var` est alors une variable de type entier. Le **type** d'une variable caractérise la nature de sa valeur.
Une affectation de variable est **une instruction**, elle diffère d'une expression car elle peut modifier l'état courant du programme.
Contrairement à d'autres langages (Java,C), en Python, le type d'une variable est dynamique, il peut changer selon sa valeur. Ainsi après l'instruction `var = 'Hello'` , la variable `var` serait de type chaîne de caractères.

C'est pourquoi, en Python, l'étape de déclaration de variable n'est pas nécessaire, elle est réalisée lors de chaque affectation.
La case mémoire associée au nom de variable est repérée par **un identifiant mémoire** obtenu avec la commande `id(var)`.
Lorsqu'il exécute une affectation de variable, l'interpréteur Python évalue d'abord **l'expression** à droite du symbole =, il remplit une case de la mémoire de l'ordinateur avec la valeur obtenue dans l'état courant de l'exécution du programme puis il associe cette case mémoire avec le nom de la variable.
Une **expression** peut être :

- une variable comme `var` ou un calcul avec des constantes comme `2+3`
- formée de plusieurs sous- expressions reliées par des opérateurs comme `2*var-5`.

Par exemple l'instruction `bar = var` crée une variable `bar` qui possède la même valeur que la variable `var`. `bar` et `var` pointeront vers la même case mémoire, on dit que `bar` est un **alias** de la variable `var`.
L'instruction `var = var+1` évalue l'expression `var+1` dans l'état courant du programme ce qui donne $2+1=3$ , remplit une case mémoire avec cette valeur et relie le nom `var` à cette case.
Si la valeur de `var+1` était déjà en mémoire, Python crée un lien entre cette case mémoire et `var`, sinon il crée une nouvelle case mémoire. Ainsi l'identifiant mémoire de la variable `var` est changé par l'affectation `var = var+1`. Mais attention, Python ne recalcule pas les valeurs des variables qui avaient été définies précédemment par une expression avec `var`.
Pour être plus précis, les données des types de base (`int`,`float`,`bool`,`string`) ne sont pas **mutables** donc elles peuvent être partagées par plusieurs variables, alors que les données de type composés (`list`,`tuple`,`dict`) sont mutables et donc non partagées (sauf si on crée un nouvel alias vers la donnée).

```
>>> var = 2
>>> id(var)
137456656
>>> bar = var #bar est un nouvel alias vers la valeur 2
>>> id(bar)
137456656
>>> mar = 3
>>> id(mar)
137456672
>>> var = var+1 #réaffectation de var, son identifiant
      change
>>> id(var),id(mar) #var est désormais un alias vers 3 la
      valeur de mar
(137456672, 137456672)
>>> var,bar,id(bar) #bar n'a pas été modifié
(3, 2, 137456656)
>>> var = var+1 #nouvelle valeur de var qui n'était pas en
      mémoire
>>> var,id(var) #un identifiant mémoire tout neuf
(4, 137456688)
>>> liste1 = [1,2,3]
>>> liste2 = [1,2,3] #liste1 et liste2 partagent la m\^eme
      liste
>>> id(liste1),id(liste2) #mais n'ont pas le m\^eme
      identifiant mémoire
(3073910700, 3073899372)
>>> liste3 = liste2 #liste3 est un nouvel alias vers la
      valeur de liste2
>>> id(liste3)
3073899372
```



**Exercice 1**

1. Quelles sont les valeurs de $x$ et $y$ après les instructions suivantes ? Prédire, puis vérifier.

```
x = 42;   y = 10;   x = y;   y = x
```

2. Mêmes questions pour :

```
x = 42;   y = 10;   z = x;   x = y;   y = z
```

3. Mêmes questions pour :

```
x = 42;   y = 10;   (x,y) = (y,x)
```

# 4 Type d'une variable

Exécuter le code suivant :

```
>>> flottant = 1/3; chaine = 'ISN'; booleen = 1>0; Booleen = 1 == 0
>>> type(flottant), type(chaine), type(booleen), type(Booleen) #D'autres types de variables
(<class 'float'>, <class 'str'>, <class 'bool'>, <class 'bool'>)
>>> id(booleen), id(Booleen) #Les nomns de variables sont sensibles à la casse
(137231552, 137231568)
>>> flottant+chaine
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

**Définition 3**

Le **type** d'une variable (et plus généralement d'une expression) indique la nature des données que cette variable contient : nombre entier,nombre à virgule, chaîne de caractères, booleens, tableaux, listes, dictionnaires ...

**Propriété 1**

Dans certains langages (C, Java ...), le type d'une variable doit être déclaré avant son affectation et il ne peut pas changer,on parle de **typage statique**.
Python est un langage à **typage dynamique**, c'est-à-dire que le type d'une variable peut changer après réaffectation. Cependant c'est un typage fort, par exemple on ne peut additionner une variable de type `float` et une variable de type `string`. En Python,comme dans la plupart des langages, les principaux types de base sont :

- le **type entier** `int`, réservé à une variable dont la valeur est un entier relatif stocké en valeur exacte. Il n'y a pas de limite de taille.
- le **type flottant** `float` pour un nombre à virgule, comme 3.14, stocké en valeur approchée sous la forme d'un triplet $(s,m,e)$ sur 64 bits : 1 bit de signe $s$, 11 bits d'exposant $e$, 52 bits de mantisse $m$.
- le **type booleen** `bool`, pour les expressions logiques comme $1<2$ qui peuvent prendre deux valeurs : vrai (`True`) ou faux (`False`).
- le type `string` pour les **chaînes de caractères** notées entre quotes simples (apostrophes) ou doubles (guillemets). Dans les chaînes entre triples quotes, on peut faire des sauts de ligne. Les primitives (fonctions définies par défaut) `int()`, `float()`, `bool()`, `str()` permettent de modifier le type d'une variable.

Exécuter le code suivant :

```
>>> booleen, int(booleen), 4/3, int(4/3)
(True, 1, 1.3333333333333333, 1)
>>> N = input('Entrez un entier : ') #La primitive input permet de capturer une entrée clavier
Entrez un entier : 12
>>> N,type(N)
('12', <class 'str'>)
>>> help(input) #La primitive help() permet l'affichage de la documentation
Help on built-in function input in module builtins:
input(...)
    input([prompt]) -> string
>>> N = int(N); type(N) #input() retourne un string qu'on change en entier avec int()
<class 'int'>
>>> import math #On importe le module avec les fonctions mathématiques (puis faire help(math) )
>>> 0.8*2**(-3),0.1*3-0.3 #Vérification (notez que l'exponentiation ** est prioritaire sur la multiplication *) et attention les flottants sont stockés en valeurs approchées
(0.1, 5.551115123125783e-17)
#si un calcul donne un flottant, utiliser round() pour arrondir
#le transtypage par int() retourne la troncature
#la division entière // retourne la partie entière du quotient (pour les entiers positifs)
>>> 5/3,round(5/3),5//3,int(5/3)
(1.6666666666666667, 2, 1, 1)
>>> -5/3,round(-5/3),-5//3,int(-5/3)
(-1.6666666666666667, -2, -2, -1)
>>> chaine1 = "possibilité d'apostrophes"; chaine2 = 'possibilité de "guillemets"' #Simples, doubles ou triples quotes pour les chaînes
>>> chaine3 = '''Un saut de ligne
... est possible'''
>>> print(chaine3);print();print(chaine1,4**2,chaine2,sep=';') #print() permet d'afficher la valeur d'une ou plusieurs variables
Un saut de ligne
est possible

possibilité d'apostrophes;16;possibilité de "guillemets"
```

**Exercice 2** *Retour sur l'affectation parallèle*

Lorsqu'il rencontre le symbole d'assignation =, Python évalue d'abord l'expression à droite puis l'assigne au nom (ou identificateur) de variable à gauche.
L'expression à droite peut être un tuple et Python peut l'affecter à un tuple d'identificateurs à gauche. On peut parler d'affectation parallèle.
Le module `dis` permet de désassembler le `bytecode` utilisé par l'interpréteur Python en une sorte de langage d'assemblage et permet de mieux comprendre le fonctionnement de l'interpréteur :

```
>>> import dis
>>> dis.dis('a = 2; b = 3; a, b = b, a')
  1           0 LOAD_CONST               0 (2)
              3 STORE_NAME               0 (a)
              6 LOAD_CONST               1 (3)
              9 STORE_NAME               1 (b)
             12 LOAD_NAME                1 (b)
             15 LOAD_NAME                0 (a)
             18 ROT_TWO
             19 STORE_NAME               0 (a)
             22 STORE_NAME               1 (b)
             25 LOAD_CONST               2 (None)
             28 RETURN_VALUE
```

L'affectation parallèle est pratique pour échanger les contenus de deux variables `x` et `y` sans variable de stockage ou pour faire des permutations circulaires. On en profite pour vérifier que Python est un langage à typage dynamique.

```
>>> a, b, c = True, 'Hello', 1
>>> a, b, c
(True, 'Hello', 1)
>>> a, b, c = c, a, b
>>> a, b, c
(1, True, 'Hello')
```

1. Ecrire un script Python qui réalise une permutation circulaire des contenus de trois variables sans affectation parallèle et avec une seule variable de stockage.
2. Ecrire un script Python qui réalise une permutation des contenus de deux variables de même type sans affectation parallèle et sans variable de stockage. Tester pour `x, y = 4, 5`. Les types des valeurs doivent être conservés. Adapter pour la permutation circulaire de trois variables.

# 5 Structures conditionnelles

**Définition 4**

Une structure conditionnelle est composée d'une d'instruction de contrôle puis d'un bloc d'instructions. En Python, l'instruction de contrôle commence par le mot clef `if` suivi d'une condition à valeur booleenne (True ou False) et se termine par le symbole « : ».
Le bloc d'instructions qui suit s'exécute si et seulement si la condition de l'instruction de contrôle a pour valeur `True`. En Python il est délimité par le nombre d'espaces en début de ligne, son indentation, qui doit être supérieure à celle du `if`.
Si la valeur de la condition est `False`, on peut proposer l'exécution d'un bloc d'instruction alternatif après le mot clef `else` suivi de « : » .
Dans d'autres langages (C, Java . . . ), les blocs d'instruction sont délimités par des accolades.

```
if condition:
    Bloc si condition == True
else:
    Bloc si condition == False
```

Syntaxe Python

**Exemple 1**

1. Le programme `if_else.py` permet de calculer l'image d'un réel $x$ par la fonction $f(x) = \begin{cases} \sin x \text{ si } -\pi \leqslant x \leqslant \pi \\ 0 \text{ sinon} \end{cases}$
   - La ligne 1 est un commentaire spécial pour indiquer à l'interpréteur l'encodage de caractères utilisé dans le script.
   - A la ligne 2 on importe toutes les fonctions du module `math` pour utiliser la fonction `sin`.
   - Expliquer l'utilisation des primitives `float()` et `input()` à la ligne 3.
   - La ligne 4 est l'instruction de contrôle composée du mot clef `if`, d'une condition `x>=-pi and x<=pi` qui prend une valeur booleenne `True` ou `False` et du signe deux points : qui termine la condition.
   - Les lignes 5 et 6 constituent le bloc d'instruction qui s'exécute si la valeur de la condition `x>=-pi and x<=pi` est **vraie** (`True`). **Toutes les instructions du bloc ont la même indentation** .
     A la ligne 6, l'opérateur modulo % permet insérer la valeur de la variable `res` dans la chaîne de caractères.
   - La ligne 7 comprend le mot clef `else` au même niveau d'indentation que le `if` et se termine par le symbole deux points :.
   - Les lignes 8 et 9 constituent le bloc d'instruction qui s'exécute si la valeur de la condition `x>=-pi and x<=pi` est `False`.
   - La ligne 10 est au même niveau d'indentation que le `if`, on est sorti du bloc d'instructions conditionnelles.
2. Dans le programme `if_else_elif.py` on a rajouté un bloc `elif` (contraction de `else if`) pour offrir trois alternatives de traitement conditionnel. `if`, `elif` et `else` sont sur le même niveau d'indentation.
   Le mot clef `pass` permet de sortir d'un bloc d'instruction.

```python
#-*- coding: Utf-8 -*
from math import*
x = float(input('Entrez un réel : \n'
    ))
if x>=-pi and x<=pi:
   res = sin(x)
   print('Le résultat est %d'%res)
else:
   res = 0
   print('Le résultat est %d'%res)
print('Les tests sont terminés')
```

if_else.py

```python
#-*- coding: Utf-8 -*
n = int(input('Entrez un entier : \n'
    ))
r = n%3
if r == 0:
   pass
elif r == 1:
   print('Gain de %d euro'%r)
else:
   print('Perte de %d euro'%r)
```

if_else_elif.py

**Remarque 2**

*Dorénavant on prendra l'habitude d'insérer dans nos programmes des bouts de codes non interprétés par l'interpréteur qui expliquent le rôle des différentes parties du programme.*
*Ces commentaires permettent à un programmeur de comprendre un programme écrit par un autre ou de relire un de ses programmes.*
*En Python tout ce qui suit le caractère dièse # jusqu'au prochain saut de ligne est un commentaire.*

**Exercice 3**

1. Ouvrir le programme `if_else.py` avec un éditeur de texte et le modifier pour qu'il prenne en entrée un réel $x$ et qu'il affiche en sortie sa valeur absolue : $|x| = \begin{cases} x \text{ si } x \geqslant 0 \\ -x \text{ si } x < 0 \end{cases}$.
2. Ouvrir le programme `if_else_elif.py` avec un éditeur de texte et le modifier pour qu'il prenne en entrée trois réels $a$, $b$ et $c$ et qu'il affiche en sortie une phrase indiquant le nombre et les valeurs des racines de l'équation $ax^2+bx+c=0$.

**Principaux opérateurs de comparaison de variables**

| | |
|---|---|
| `x == y` | `x est égal à y` |
| `x != y` | `x est différent de y` |
| `x > y` | `x est strictement supérieur à y` |
| `x < y` | `x est strictement inférieur à y` |
| `x >= y` | `x est supérieur ou égal à y` |
| `x <= y` | `x est inférieur ou égal à y` |
| `x is y` | `id(x) == id(y)` |

**Principaux opérateurs sur des expressions booleenes**

| | |
|---|---|
| `E and F` | `Vraie si E est Vraie ET F est Vraie` |
| `E or F` | `Vraie si E est Vraie OU F est Vraie` |
| `not E` | `Vraie si E est Fausse` |

**Exercice 4** *Importance de l'indentation*

Dans l'éditeur, saisir puis sauver/exécuter les scripts suivants :

```python
#script 1
if 10**3 < 999:
    print("Ga")

if 6**2 > 30:
    print("Bu")

if 10**3 < 999 or 6**2
       > 30:
    print("Zo")
```

```python
#script 2
if 10**3 < 999:
    print("Ga")
    if 6**2 > 30:
        print("Bu")

if 10**2 < 99 or 5**2
      > 20:
    print("Zo")
```

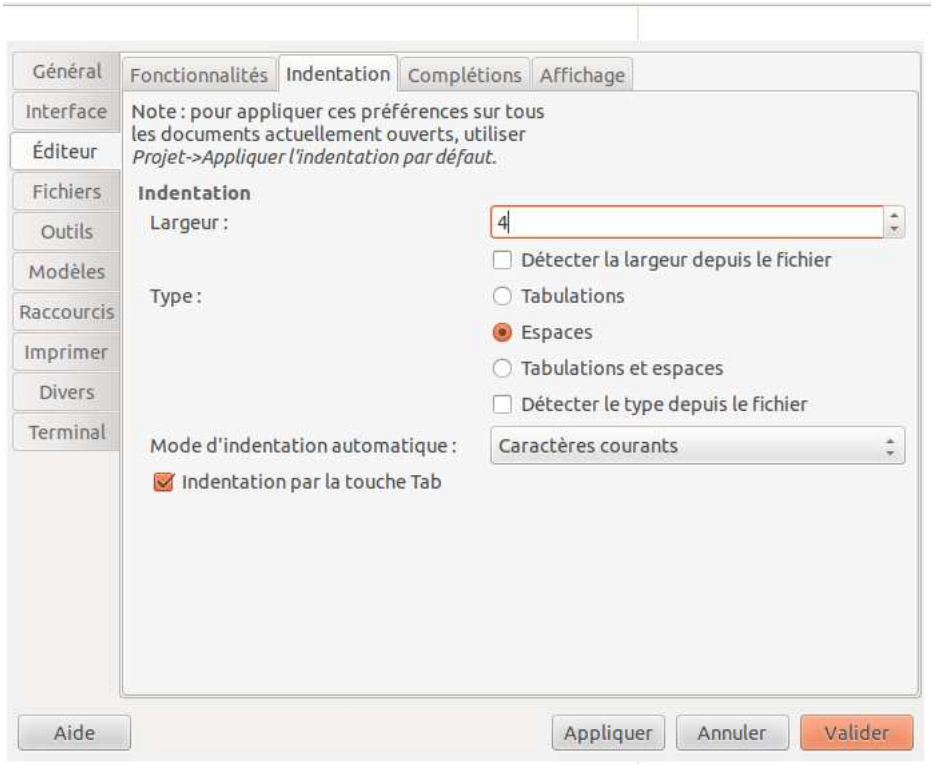
```python
#script 3
if 10**3 < 999:
    print("Ga")
    if 6**2 > 30:
        print("Bu")
    if 10**3 < 999 or
          6**2 > 30:
        print("Zo")
```

**Remarque 3**

***En Python l'indentation recommandée est de 4 espaces.***

*En pratique, dans les éditeurs spécialisés pour Python (`Idle`, `Spyder`), l'indentation est réglée automatiquement sur 4 espaces, la touche de tabulation correspond à une indentation et les tabulations sont automatiquement remplacées par des espaces (4 en l'occurrence).*

*Nous utiliserons* `Geany`*, un éditeur de texte généraliste, donc il nous faudra d'abord paramétrer l'éditeur comme ci-dessous dans la fenêtre accessible depuis le menu* `Préférences>Editer` *ou le raccourci clavier* `CTRL+ALT+P`*.*




***Il faut se méfier de la tabulation****. C'est un caractère invisible comme l'espace mais dont la largeur varie selon les éditeurs de texte. Si on mélange tabulations et espaces, un code peut sembler bien indenté dans un éditeur mais ce ne sera pas le cas dans un autre et l'interpréteur Python va générer des erreurs.*
*Une fois qu'on a fait les réglages ci-dessus dans* `Geany`*, on est tranquille et toutes les tabulations seront transformées en espace. Pour s'en assurer on peut activer l'option* `Affichage>Editeur>Afficher les espaces`*. Les espaces s'affiche avec des . et les tabulations avec des flèches. On peut aussi demander la conversion de toutes les tabulations en espaces avec* `Document>Remplacer les tabulations par des espaces`*.*

***Pour plus d'informations sur les bonnes pratiques de rédaction d'un code Python, on pourra consulter la PEP 8 :***
`http://www.python.org/dev/peps/pep-0008/`

*Par ailleurs il est bon ce connaître quelques raccourcis clavier valables dans la plupart des éditeurs de texte :*

- `CTRL+A` *pour tout sélectionner*
- `CTRL+C` *pour copier la sélection*
- `CTRL+V` *pour coller la sélection*
- `CTRL+X` *pour couper la sélection*
- `CTRL+S` *pour enregistrer le fichier ouvert*
- `CTRL+I` *pour indenter la sélection*
- `CTRL+U` *pour désindenter la sélection*
- `CTRL+Z` *pour annuler l'action précédente*

*Enfin, on peut remarquer que si on enregistre notre code source dans un fichier d'extension* `.py`*,* `Geany` *active* ***la coloration syntaxique*** *et* ***l'auto-complétion*** *pour Python, comme les éditeurs spécialisés (*`Idle`*,* `Spyder`*).*

- ***la coloration syntaxique*** *met en évidence les mots clefs du langage par un code de couleurs*
- ***l'auto-complétion*** *affiche une info-bulle avec une liste de fonctions du langage qui commence par les mêmes caractères que ceux qu'on vient de taper.*

**Méthode** *Importer un module*

On a parfois besoin d'utiliser des fonctions de Python qui ne sont pas chargées pas défaut. Ces fonctions sont stockées dans des programmes Python appelées **modules** ou **bibliothèques**. Par exemple le module `math` contient les fonctions mathématiques usuelles et le module `random` contient plusieurs types de générateurs de nombres pseudo-aléatoires.
Pour importer une fonction d'un module on peut procéder de deux façons :

```
#import du module de mathématique (création d'un point d'accès)
import math

#pour utiliser la fonction sqrt, on la préfixe du nom du module et d'un point
racine = math.sqrt(2)
```

Première façon

```
#import de la fonction sqrt du module math
from math import sqrt

racine = sqrt(2)

#Pour importer toutes les fonctions de math, ecrire
#from math import *
```

Deuxième façon

Pour obtenir de l'aide sur le module math dans la console Python, il faut d'abord l'importer avec `import math` puis taper `help(math)`, mais le mieux est encore de consulter la documentation sur internet `http://docs.python.org/3/`.

Principales fonctions du modules `random` :

| Fonction | Effet |
|---|---|
| `randrange(a,b)` | retourne un entier aléatoire dans [a ;b[ |
| `randint(a,b)` | retourne un entier aléatoire dans [a ;b] |
| `random()` | retourne un flottant aléatoire dans [0 ;1[ |
| `uniform(a,b)` | retourne un flottant aléatoire dans [a ;b] |

**Exercice 5**

L'Indice de Masse Corporelle se calcule par la formule IMC = $\frac{masse}{taille^2}$ où la masse est en kilogrammes et la taille en mètres. Un IMC est considéré comme normal s'il est compris entre 18,5 et 25. En dessous de 18, la personne est en sous-poids et au-dessus de 25 elle est en surpoids.

1. Ecrire un programme qui demande la taille et retourne l'intervalle de masse correspondant à un IMC normal.
2. Ecrire un programme qui demande la masse et retourne l'intervalle de taille correspondant à un IMC normal.

**Exercice 6**

Les années bissextiles sont les années non séculaires divisibles par 4 ou les années séculaires divisibles par 400.
Ecrire un programme qui prend en entrée une année $n$ et qui affiche en sortie si elle est bissextile.

# 6 Boucles

## 6.1 Boucles `for`

**Définition 5** *Boucle `for` ou `Pour`*

Si on veut répéter un bloc d'instructions un nombre déterminé `n` de fois, on utiliser une boucle `for` avec une variable compteur `i` qui va prendre successivement les valeurs de `m` à `n` avec un pas de `s`.
En Python, la fonction `range` crée un itérateur, qui est une sorte de distributeurs d'entiers consécutifs.

- `range(n)` retourne un itérateur parcourant les entiers consécutifs entre 0 et `n` exclu.
- `range(m,n)` retourne un itérateur parcourant les entiers consécutifs entre `m` compris et `n` exclu.
- `range(m,n,s)` retourne un itérateur parcourant les entiers consécutifs entre `m` compris et `n` exclu avec un pas de `s`..

En plus de `break`, Python propose le mot clef `continue` qui permet dans une boucle `Pour` de passer directement à l'itération suivante.

```
for i in range(m,n,s):
    Bloc

for i in range(n):
    #i varie de 0 à n-1
    Bloc
```

Syntaxes Python

**Exercice 7**

Dans l'éditeur, saisir puis sauver/exécuter les lignes suivantes :

```
for i in range(10,15):
    print("bonjour ",i)
```

Remplacer `print("bonjour ",i)` par `print("bonjour "+i)`, exécuter.
Remplacer enfin `print("bonjour ",i)` par `print("bonjour "+str(i))` et exécuter.

**Exercice 8** *Applications directes*

1. Ecrire un programme qui prend en entrée un entier $n$ et qui retourne les vingt premiers multiples entiers de $n$.
2. Ecrire un programme qui prend en entrée un entier $n$ et qui retourne la valeur de la somme $\sum_{k=1}^{n} k$.
3. Ecrire un programme qui prend en entrée un entier $n$ et qui retourne la valeur du terme de rang $n$ de la suite $(u_n)$ définie par :
$$u_0 = 2 \quad \text{et} \quad u_{n+1} = \frac{1}{2}u_n + \frac{1}{2}n + 1$$
4. Ecrire un programme qui prend en entrée un entier $n$ et qui affiche tous les entiers pairs inférieurs ou égaux à $n$.
5. Ecrire un programme qui affiche la chaîne de caractères `'100+99+98+97+...+1'` (somme des 100 premiers entiers dans l'ordre décroissant sous forme de chaîne de caractères).
6. Ecrire un programme qui affiche tous les couples $(i, j)$ avec $1 \leqslant i \leqslant 6$ et $1 \leqslant j \leqslant 6$ comme ci-dessous :

```
(1, 1),(1, 2),(1, 3),(1, 4),(1, 5),(1, 6)
(2, 1),(2, 2),(2, 3),(2, 4),(2, 5),(2, 6)
(3, 1),(3, 2),(3, 3),(3, 4),(3, 5),(3, 6)
(4, 1),(4, 2),(4, 3),(4, 4),(4, 5),(4, 6)
(5, 1),(5, 2),(5, 3),(5, 4),(5, 5),(5, 6)
(6, 1),(6, 2),(6, 3),(6, 4),(6, 5),(6, 6)
```

## 6.2 Boucles `while`

**Définition 6** ***Boucle `while` ou `TANT QUE`***

C'est une structure itérative dont la première instruction est `while condition:` suivie d'un bloc d'instruction d'indentation supérieure qui s'exécute tant que la valeur booleenne de la condition est vraie (`True`). Si cette valeur est `False` la boucle est achevée.
***Attention au risque de boucle infinie avec les boucles `while` si le test d'arrêt n'est jamais vérifié ...***

**Exemple 2**

La boucle 1 ci-contre calcule le logarithme entier d'un réel $x$ supérieur ou égal à 1, c'est-à-dire le nombre de fois qu'il faut le diviser par deux pour obtenir un nombre inférieur ou égal à 1. Ainsi le logarithme entier de 1 000 est 10 car $\frac{1000}{2^{10}} \leqslant 1$ et $\frac{1000}{2^9} > 1$.
La condition de sortie de la boucle 2 est contrôlée par l'utilisateur, s'il ne répond pas 'y', la boucle sera infinie.

```
#boucle 1
x = float(input('Entrez un réel : \n'))
i, stock = 0, x
while x>1:
    x = x/2
    i += 1
print('Le logarithme entier de %s est %s'%(
     stock,i))

#boucle 2
test = True
while test:
    rep = input('Sortie de boucle ? (y/n)')
    if rep == 'y':
        test = False
```

**Exercice 9**

Ecrire un programme qui demande de parier sur le nombre de coups nécessaires pour obtenir un 6.
Le programme lance un dé à six faces jusqu'à l'obtention du 6 et affiche si l'utilisateur a remporté son pari.

**Exercice 10**

1. Ecrire un programme qui prend en entrée un réel $M$ et qui affiche en sortie le plus petit entier $n$ tel que $\sum_{k=1}^{n} k^3 > M$.
2. Écrire un programme où l'ordinateur choisit un mot de passe puis demande à l'utilisateur de saisir le mot de passe tant que celui-ci n'est pas correct.
3. On considère la suite $(u_n)$ définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = 2u_n + n \end{cases}$.
   Ecrire un programme qui retourne le plus petit indice $n$ tel que $u_n > 10000$. Modifier ce programme pour qu'il affiche le plus grand indice $n$ tel que $u_n \leqslant 10^6$.

**Exercice 11** *Rurple épisode 1*

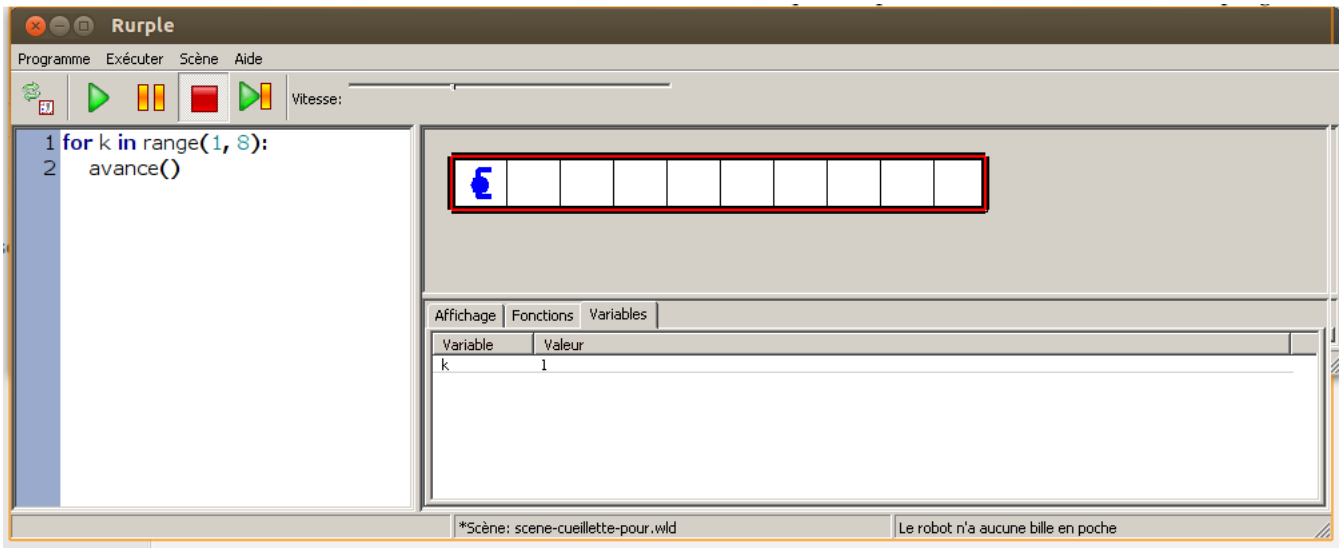
Lancer le logiciel Rurple de puis le partage réseau `T`. Ce logiciel permet de déplacer un robot nommée Rurple sur une scène en utilisant le langage de programmation Python augmenté de quelques fonctions spécifiques.
Voici les principales fonctions qui permettent de déplacer le robot pu de la faire interagir avec son environnement :

- « `avance( )` » (ou appui sur touche A) : le robot avance d'une case devant lui.
- « `gauche( )` » (ou appui sur touche G) : le robot effectue un quart de tour vers la gauche.
- « `depose( )` » (ou appui sur touche D) : le robot dépose une bille sur la case où il se trouve.
- « `prends( )` » (ou appui sur touche P) : le robot ramasse une bille sur la case où il se trouve.

L'interface graphique est composé d'une barre de menu classique en haut et de trois fenêtres :

- La fenêtre de gauche accueille le programme (ou script) des *instructions* que l'on souhaite faire exécuter au robot. Celui-ci lit le programme de haut en base et exécute les instruc*tions de façon séquentielle. Certaines instructions comme les conditions ou les boucles* permettent de faire des sauts en avant ou en arrière dans le *code source* du programme. On peut enregistrer/ouvrir/créer un nouveau un programme/script (au format *.py comme un script Python) en sélectionnant le menu Programme. Il faut enregistrer un nouveau programme avant de l'exécuter.
- La fenêtre en haut à droite contient la grille d'évolution du robot, on peut régler sa taille à partir du menu Scène et il est possible d'insérer des murs ou des billes dans la grille en cliquant dessus avec le bouton gauche. On peut ouvrir/créer une nouvelle/enregistrer (au format *.wld) une scène depuis le menu Scène. Pour réinitialiser la scène, on peut cliquer sur le bouton correspondant de la barre d'outils ou effectuer la combinaison de touches sur `CTRL + R`.
- La fenêtre en bas à droite contient trois volets Affichage/Fonctions/Variable. Dans Affichage seront écrites les messages du robot (c'est le programmeur qui le fait parler ...). Dans Fonctions se trouvent la liste des fonctions spécifiques à Rurple en plus des fonctions classiques de Python. Dans Variables on pourra observer l'évolution du contenu des variables.

En dessous de la barre de menu se trouve un barre d'outils avec trois icônes cliquables pour contrôler l'exécution d'un programme (lecture puis interprétation par le robot) selon trois fonctionnalités : Exécuter, Suspendre, Stopper, Exécuter pas à pas (fait une pause après chaque ligne/instruction).




1. Ecrire une boucle `for` qui permet au robot de parcourir de gauche à droite toute la scène de dimension $8 \times 1$ ci-dessus.
   Le programme fonctionne-t-il si on change la largeur de la scène ?
2. Créer un nouveau programme (`boucletantque.py`), saisir le code 1 ci-contre avec une boucle `while` pour laquelle on retrouve les deux points et l'indentation. Exécuter le programme avec une scène rectangulaire $8 \times 1$ comme pour la boucle `for` ci-dessus.
3. Observer l'évolution de la variable de boucle `k` en exécutant le programme pas à pas.
4. Que se passe-t-il si on change la largeur de la scène ? Tester pour une largeur inférieure puis supérieure à 8.
5. Créer un nouveau programme (`boucletantque2.py`) avec le code 2 ci-contre. Tester ce programme. Quel avantage présente-t-il ?
6. Si on remplace l'instruction `avance()` par `print('Bonjour')` que se passe-t-il ?

```
#code tant que 1
k = 0
while k < 7 :
    k = k + 1
    avance()
```

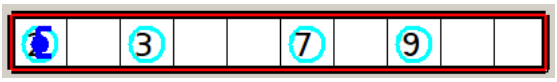
```
#code tant que 2
while not mur_devant():
    avance()
```

**Exercice 12** *Rurple épisode 2*

1. En utilisant une boucle Tant Que, créer un nouveau programme `cueillettetantque.py` qui permet au robot de cueillir toutes les fraises déposées sur une scène rectangulaire de dimensions $n \times 1$ avec au plus 1 fraise par case.

2. En utilisant une boucle Tant Que, créer un nouveau programme `cueillettetantque1.py` qui permet au robot d'avancer puis de cueillir toutes les fraises déposées sur la deuxième case comme ci-dessous où il y a 3 fraises.


3. En utilisant une boucle Tant Que, créer un nouveau programme `cueillettetantque2.py` qui permet au robot d'avancer puis de cueillir toutes les fraises déposées sur les cases d'une scène rectangulaire de dimensions $n \times 1$ avec éventuellement plus d'une fraise par case.



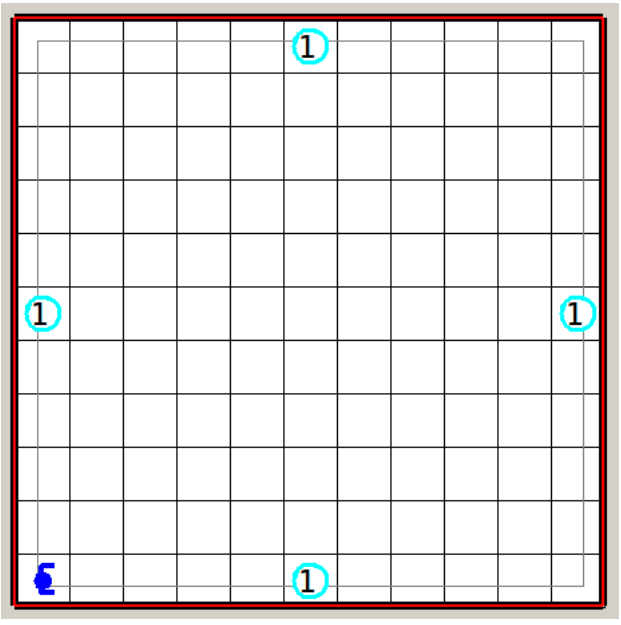

**Exercice 13** *Rurple épisode 3*

1. En utilisant deux boucles `for` imbriquées, écrire un programme de quatre lignes qui permet au robot de faire le tour d'une scène de dimensions $4 \times 4$ et de revenir à sa position de départ comme dans l'exemple initial.

2. Comment modifier le programme précédent pour que le robot puisse faire le tour de n'importe quelle scène rectangulaire de dimensions $n \times m$ avec $n > 1$ et $m > 1$ ?

3. Dans le menu scène, sélectionner Nombre de billes et placer quatre billes dans la poche du robot.
Modifier le programme précédent pour que le robot fasse un tour de la scène et dépose une bille à chaque coin de n'importe quelle scène rectangulaire.

4. Choisir une scène rectangulaire de dimensions $11 \times 11$. Modifier le programme précédent pour que le robot fasse un tour de la scène et dépose une bille au milieu de chaque côté.



## Exercice 14

1. **a.** Soit $i$ un entier, développer $(i+1)^2$.

   **b.** Sans utiliser l'opérateur d'exponentiation $**$, compléter le script Python ci-contre pour que la variable `s` contienne en sortie de boucle la somme $\sum_{i=0}^{n} i^2$.

2. Un entier naturel supérieur ou égal à 2 est premier s'il n'est divisible que par 1 et par lui-même.
   Une propriété (à justifier) établit qu'un entier $n$ est premier s'il n'est divisible par aucun des entiers compris entre 2 et $\lfloor\sqrt{n}\rfloor$ (la partie entière de $\sqrt{n}$).
   Ecrire un programme qui détermine si un entier entré par l'utilisateur est premier.

```
n = int(input('Entrez un entier n : '))
i = 0
c = i
s = c
while i<n:
    c = ......
    s = ......
    i = ......
print('somme des carrés de 0 à ',n,' : '
      ,s)
```

## Exercice 15

1. Exécuter le script ci-contre pour $a = 230$ et $b = 113$ et représenter l'évolution des variables dans un tableau.

2. Que représentent les valeurs des variables `m` et `n` affichées en sortie ?

```
a = int(input('a='))
b = int(input('b='))
if a > 0 and b > 0:
    m, n = a, 0
    while m >= b:
        m, n = m - b, n + 1
print('m=',m,'n=',n)
```

## Exercice 16

1. Ecrire avec une boucle `while` un programme qui calcule le quotient de $a$ par $b$ où $a$ et $b$ sont des entiers naturels saisis au clavier.

2. Faire une recherche sur internet sur les différents algorithmes d'Euclide qui permettent de calculer le PGCD de deux entiers naturels $a$ et $b$.

   **a.** Programmer avec une boucle `while` l'algorithme dit « des divisions successives ».

   **b.** Programmer avec une boucle `while` l'algorithme dit « des différences successives ».

**Méthode** *Boucle infinie avec break*

Parfois (dans les jeux), on utilise des boucles infinies qui tournent tant qu'un événement particulier ne s'est pas produit. En Python, le mot clef `break` permet de sortir d'une boucle `while` directement à partir du corps de boucle.

```
password = 'secret'
while True:
    entree = input('Entrez le mot de passe')
    if entree == password:
        break
```

**Exercice 17** *Terminaison de boucle*

1. Parmi les programmes `boucle1` et `boucle2` ci-contre déterminer ceux qui comportent une boucle qui ne se termine pas.

2. Une suite $(u_n)$ est dite de Syracuse si elle est définie ainsi :

$$\begin{cases} u_0 \text{ est un entier positif} \\ u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ si } u_n \text{ est pair} \\ 3u_n + 1 \text{ si } u_n \text{ est impair} \end{cases} \end{cases}$$

Une conjecture non démontrée stipule que pour tout entier positif $u_0$ la suite de Syracuse $(u_n)$ atteint la valeur 1 pour un certain indice.

Ecrire un programme qui demande d'entrer un entier $a$ et qui retourne l'indice du premier terme égal à 1 pour la suite de Syracuse de premier terme $u_0 = a$.

```
p = False
n = 0
while not p:
    n+=1
    if n**2 == 81:
        p = True
print(n)
```

boucle1.py

```
from math import*
p = True
n = 0
while p:
    n+=1
    if cos(n) == 0:
        p = False
print(n)
```

boucle2.py

## 6.3 Savoir tester et instrumenter un programme

**Méthode**

1. Pour déterminer si un programme donne les résultats attendus correctement il faut l'exécuter plusieurs fois avec un **jeu d'entrées tests** telles que :
   - on connaît à l'avance le résultat pour ces entrées tests ;
   - tous les cas possibles sont recouverts par le jeu de tests même les cas limites.

2. Si un programme ne fonctionne pas correctement, pour détecter les erreurs, on peut l'**instrumenter**, c'est-à-dire ajouter des instructions de sortie dans le programme (avec des `print`), qui permettent de visualiser ce qu'il se passe au cours de son exécution. Pour des programmes longs, on peut utiliser un outil spécialisé comme un débogueur.
   Pour de gros programmes, le jeu de tests et 'instrumentation doivent être prévus dès le début, ce n'est pas une perte de temps. Une méthode rudimentaire sans débogueur peut consister à créer une variable `debug` de type booléen et d'insérer des instructions de débogage (par exemple pour le suivi des variables) dans des blocs conditionnels commandés par la valeur de `debug`. On donne ci-dessous un exemple de débogage d'un script correct (pourquoi ?) de calcul de la somme des chiffres d'un entier écrit en base 10 :

```
debug = True
if debug:
    k = 0
n = int(input('Entrez un entier : '))
somme = 0
while n > 0:
    somme = somme + n%10
    if debug:
        k += 1
        print('itération :', k)
        print('Somme : ', somme)
        print('n : ', n, end='\n\n')
    n = n // 10
```

```
Entrez un entier : 123
itération : 1
Somme : 3
n :  123

itération : 2
Somme : 5
n :  12

itération : 3
Somme : 6
n :  1

6
```

**Exercice 18**

1. Proposer un jeu de tests satisfaisant pour le programme de calcul des solutions réelles d'une équation du second degré.
2. Le programme ci-contre a pour but de calculer la somme des carrés des entiers successifs entre 1 et $n$.
   a. Tester ce programme pour déterminer s'il est correct.
   b. Instrumenter ce programme pour déterminer l'erreur.

```
n = int(input('Entrez un entier'))
i = 1
s =0
while i<=n:
    i = i*i
    s = s+i
    i= i+1
print(s)
```

## 6.4 Des boucles simples et des doubles

**Exercice 19**

1. Ecrire un programme qui affiche les vingt premiers termes de la table de multiplication par 11 à l'aide d'une boucle `for` puis modifier ce programme pour qu'il utilise une boucle `while`.
   *Les boucles `for` sont un cas particulier de boucles `while`.*
2. Ecrire une programme prend en entrée un entier naturel $n$ et qui affiche en sortie le terme de rang $n$ de la suite de Fibonacci :

$$u_n = \begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

3. Écrire un programme qui calcule et affiche la liste des diviseurs d'un nombre entier naturel entré au clavier.
4. Combien d'étoiles * sont affichées par les programmes `etoile1` et `etoile2` ci-contre qui utilisent deux boucles imbriquées ?

```
#-*- coding: Utf-8 -*-
for i in range(100):
    print('*')
    for j in range(100):
        print('*')
```

etoile1.py

```
#-*- coding: Utf-8 -*-
for i in range(100):
    for j in range(100):
        print('*')
```

etoile2.py

**Exercice 20**

1. Tester le programme suivant pour n=4, n=10, n=16.
2. Que représente la somme calculée par ce programme ?
3. Modifier ce programme pour qu'il n'utilise pas le mot clef `continue`.

```
# -*- coding: utf-8 -*-

n = int(input('Entrez un entier n : \n'))
somme = 0
for i in range(n):
    if i%2 == 0:
        continue
    if i%3 == 0:
        #equivalent de somme = somme + i**2
        somme += i**2
```

```
11 print('La somme cherchée est %s'%somme)
```

continue

**Exercice 21**

1. On considère la suite $(u_n)$ définie par $\begin{cases} u_0 = 3 \\ u_{n+1} = 3u_n + n \end{cases}$.

   Ecrire un script Python qui prend en entrée un entier $n$ et qui retourne le terme de rang $n$ de la suite $(u_n)$.

2. On admet que la suite $(u_n)$ définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = 1 + \dfrac{2}{u_n} \end{cases}$ est définie pour tout $n \in \mathbb{N}$ et converge vers 2.

   Ecrire un script qui détermine le plus petit entier $p$ tel que $|u_p - 2| < 10^{-6}$.

**Exercice 22**

Pour chaque script déterminer le nombre d'étoiles affichées :

```
#Script 1
i, j = 100, 100
while i> 0:
    i = i-1
    print('*')
    while j>0:
        j = j-1
        print('*')
```

```
#Script 2
i = 100
while i> 0:
    i = i-1
    print('*')
    j = 100
    while j>0:
        j = j-1
        print('*')
```

```
#Script 3
i = 100
while i> 0:
    i = i-1
    j = 100
    while j>0:
        j = j-1
        print('*')
```

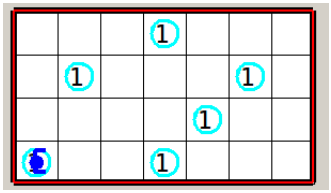
```
#Script 4
i, j = 100, 50
while i>j:
    i = i-1
    j = 50
    print('*')
    while j>0:
        j = j-1
        print('*')
```

**Exercice 23** *Rurple épisode 4 (plus difficile)*

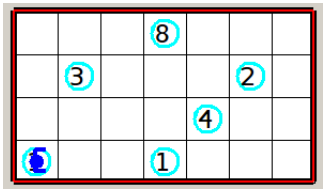
*Cet exercice pourra être repris quand nous aurons appris comment regrouper un bloc d'instructions dans une fonction. Celà permet d'éviter les copier-coller lorsqu'un bloc d'instructions est réutilisé.*

1. Quelle suite d'instructions permet au robot de faire demi-tour ? de tourner à droite ?

2. Créer un nouveau programme qui permette au robot de cueillir toutes les billes/fraises déposées sur une grille rectangulaire de dimensions $7 \times 4$ avec au plus une fraise sur chaque case (y compris sur la case de départ). Le programme doit fonctionner quelle que soit la disposition des fraises.
   A la fin de la cueillette, le robot doit afficher le nombre de fraises cueillies.



3. Modifier le programme précédent pour qu'il permette au robot de cueillir toutes les billes/fraises déposées sur n'importe quelle grille rectangulaire de dimensions $7 \times 4$ avec éventuellement plus d'une fraise sur chaque case (y compris sur la case de départ).



# 7 Dessiner avec le module `turtle`

**Exercice 24** *Dessin avec le module* `turtle`

Le module `turtle` de Python permet de construire des figures en donnant des instructions à une tortue comme dans le langage LOGO.

1. Faire une recherche sur internet sur le langage LOGO et la tortue LOGO.
2. Tester le programme suivant.
3. Modifier ce programme pour qu'il dessine un triangle équilatéral, puis un hexagone puis un octogone .....
4. Remplacer la valeur 90 par 70 en ligne 7 et insérer l'instruction `i = 0` juste avant le `while` de la ligne 5 puis les instructions `i=i+1` et `print(i)` comme premières instructions du bloc commandé par le `while`.
5. Tester le programme modifié. Que se passe-t-il ?
6. Calculer le plus petit commun multiple de 70 et 360 avec un autre programme et expliquer le déroulement du programme `polygone.py`.

*Il semblerait plus naturel de remplacer le test* `abs(pos())<1` *par* `pos() == (0,0)` *. On peut essayer mais celà ne fonctionne pas comme prévu . . . Pour stopper une boucle infinie on peut faire* `Ctrl+C`*. La fonction* `pos()` *retourne les coordonnées de la position courante de la tortue sous la forme d'un couple de flottants et contrairement aux entiers les flottants sont représentés en valeur approchée et une valeurs très proche de (0,0) n'est pas (0,0) pour l'ordinateur . . .*

```
1  from turtle import * #import de toutes les fonctions du module turtle
2  pencolor('red') #couleur du crayon
3  # la boucle while contient les instructions données à la tortue
4  pos() #affichage de la position initiale de la tortue : l'origine (0,0)
5  while True:
6      forward(100) #avance de 100 pixels
7      left(90) #tourne à gauche de 90 degrés
8      if abs(pos())<1: #si la position est à une distance de (0,0) inférieure à 1
9          break # on sort de la boucle
10 done() #pour lancer la construction
```

polygone.py

**Principales fonctions du module `turtle`**

| | |
|---|---|
| `reset()` | effacer tout et recommencer |
| `goto(x,y)` | aller au point de coordonnées (x,y) |
| `forward(d)` | avancer de d |
| `backward(d)` | reculer de d |
| `up(), down()` | lever ou baisser le crayon |
| `left(`$\theta$`), right(`$\theta$`)` | tourner à gauche et à droite de $\theta$ degrés |
| `color('red'),width(l)` | couleur, épaisseur du tracé |
| `circle(R)` | trace un cercle de rayon R |
| `done() ou mainloop()` | lance la construction |

**Exercice 25**

Tester puis expliquer les programmes suivants :

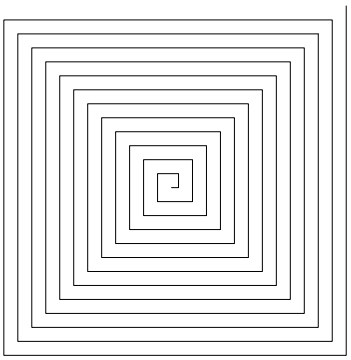
```
from turtle import*
for i in range(4):
    forward(100)
    left(90)
mainloop()
```

```
from turtle import*
rayon = 20
while rayon<100:
    circle(rayon)
    rayon=rayon+10
mainloop()
```

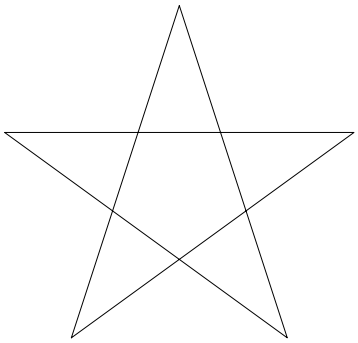
```
#-*- coding: Utf-8 -*
from turtle import*
rayon = 20
while rayon<100:
    circle(rayon)
    up()
    right(90)
    forward(10)
    left(90)
    down()
    rayon=rayon+10
mainloop()
```

1. Ecrire un programme qui permet de tracer un hexagone régulier puis plus généralement un polygone régulier à $n$ côtés (faire une rechercher internet sur la somme des angles d'un polygone régulier à $n$ côtés).
2. Ecrire un programme qui permet de tracer une spirale polygonale comme celle ci-dessous puis modifier un paramètre du programme pour tracer d'autres types de spirales polygonales.
3. Ecrire un programme qui permet de tracer une étoile à cinq branches comme celle ci-dessous.
4. Ecrire un programme qui dessine le drapeau européen avec douze étoiles à cinq branches jaunes sur un fonds bleu.

**Une spirale**



**Une étoile**



# Table des matières