AP1

Bases de programmation en Python

Code *AP1*.1: Hello World en Python

```
print("Hello World !") # Écrit "Hello World !" à l'écran
```

Code *AP1*.2: Hello World en C

```
// Exemple de programme C qui va afficher
// sur l'écran, le message "Hello world !"

#include <stdio.h>

void main(void)
{
 printf("Hello world !\n");
}
```

Code *AP1*.3: Fonction bien indentée

```
>>> def fonction_bien_indentee(x):          # Ici pas de souci
...     result = 42 - x
...     return result
...
>>> fonction_bien_indentee(12)              # Vérification
30
```

Code *AP1*.4: Oubli d'indentation...

```
>>> def fonction_non_indentee(x):          # Là, GROS souci !
... result = 42 - x                        # Des tas d'erreurs sont signalées
  File "<stdin>", line 2
    result = 42 - x                        # Des tas d'erreurs sont signalées
         ^
IndentationError: expected an indented block
>>> return result
  File "<stdin>", line 1
SyntaxError: 'return' outside function

>>> fonction_non_indentee(12)              # Ca ne marche bien évidemment pas...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fonction_non_indentee' is not defined
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fonction_non_indentee' is not defined
```

Code *AP1*.5: Mélange d'espaces et de tabulations

```
>>> def fonction_melange_tabs_espaces(x): # Celle-ci est plus dure à voir
...     result = 42-x                     # Là, on a mis 4 espaces
...     return result                     # et là un caractère de tabulation
  File "<stdin>", line 3
    return result                     # et là un caractère de tabulation
                                                                        ^
TabError: inconsistent use of tabs and spaces in indentation

>>> fonction_melange_tabs_espaces(12)     # Ca ne marche pas non plus...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fonction_melange_tabs_espaces' is not defined
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fonction_melange_tabs_espaces' is not defined
```

Code *AP1*.6: Entiers et flottants

```
>>> 1//2  # Le quotient de la division euclidienne de 1 par 2 vaut 0
0
>>> 1/2   # Mais la division "habituelle" de 1 par 2 donne 0.5,
0.5
>>>       # NB: ce n'était pas le cas avant la version 3 de Python
>>>       # => Par précaution, toujours écrire 1.0/2 ou 1/2.0 pour forcer
>>>       #    la conversion des entiers en flottants
>>> # Toutes les manips habituelles (somme, soustraction, multiplication,
>>> # division, puissance, modulo) marchent gentiment
>>> 6.25+0.5, 6.25-0.5, 6.25*0.5, 6.25/0.5, 6.25**0.5, 6.25%0.5
(6.75, 5.75, 3.125, 12.5, 2.5, 0.25)
>>> # float() et int() permettent de passer de l'un à l'autre
>>> float(1),int(6.25)
(1.0, 6)
```

Code *AP1*.7: Booléens

```
>>> t = True
>>> f = False
>>> t and t, t and f, f and t, f and f
(True, False, False, False)
>>> t or  t, t or  f, f or  t, f or  f
(True, True, True, False)
>>> x = 1
>>> not x > 2  # Est-il possible que x ne soit pas strictement supérieur à 2 ?
True
>>> x = 3
>>> not x > 2  # Est-il possible que x ne soit pas strictement supérieur à 2 ?
False
>>> # Les différents opérateurs de comparaison
>>> 1 < 1, 1 <= 1, 1 > 1, 1 >= 1, 1 != 1, 1 == 1, 1 is 1
(False, True, False, True, False, True, True)
>>> L1,L2 = [1,2,3,4],[1,2,3,4]
>>> L3 = L2
>>> L1 == L2  # L1 et L2 représentent-ils la même chose ?
True
>>> L1 != L2  # L1 et L2 représentent-ils des choses différentes ?
False
>>> L1 is L2  # L1 et L2 sont-ils identiques (même objet) ?
False
>>> L2 is L3  # L2 et L3 sont-ils identiques (même objet) ?
True
```

Code *AP1*.8: Chaîne de caractères

```
>>> # La définition d'une chaîne peut se faire entre guillemets
>>> simples = 'simples'
>>> doubles = "doubles"
>>> triples = '''ou triples qui seuls acceptent
... les sauts de ligne.'''

>>> # Quelques fonctions utiles rassemblées par usage (cf help(str) pour plus)
>>> # Remarquez l'usage des guillemets doubles " car la chaîne contient
>>> # elle-même un guillement simple (sous forme d'apostrophe)
>>> ma_chaine = "boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?"

>>> # Jeux de majuscules et minuscules
>>> ma_chaine.capitalize()  # Premiere letre en majuscule (capitals en anglais)
"Bonjour les cocos, comment allez-vous aujourd'hui ?"
>>> ma_chaine.lower()       # Tout en minuscules (lowercase en anglais)
"bonjour les cocos, comment allez-vous aujourd'hui ?"
>>> ma_chaine.swapcase()    # Inverse majuscules et minuscules
"BOnjOUR LeS cOCos, COmmENT ALlez-VouS AUJouRD'HUI ?"
>>> ma_chaine.upper()       # Tout en majuscules (uppercase en anglais)
"BONJOUR LES COCOS, COMMENT ALLEZ-VOUS AUJOURD'HUI ?"

>>> # Problèmes de justification
>>> ma_chaine.center(70)    # Centrage (sur 70 caractères de large au total)
"         boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?          "
>>> ma_chaine.ljust(70)     # Justifie à gauche sur 70 caractères au total
"boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?                   "
>>> ma_chaine.rjust(70)     # Justifie à droite sur 70 caractères au total
"                   boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?"
>>> c = ma_chaine.center(70)
>>> c.lstrip()              # Enlève tous les espaces à gauche de la chaîne
"boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?          "
>>> c.strip()               # Enlève tous les espaces  autour  de la chaîne
"boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?"
>>> c.rstrip()              # Enlève tous les espaces à droite de la chaîne
"         boNJour lEs CocOS, coMMent alLEZ-vOUs aujOUrd'hui ?"

>>> # Détection de sous-chaînes
>>> ma_chaine.count('OU')     # Compte le nombre d'occurrences dans la chaîne
2
>>> ma_chaine.endswith('i?')  # Cela finit-il ainsi ?
False
>>> ma_chaine.startswith('bo')# Cela commence-t-il ainsi ?
True

>>> # Découpage de chaîne
>>> ma_chaine.split()         # Découpage, par defaut selon les espaces
['boNJour', 'lEs', 'CocOS,', 'coMMent', 'alLEZ-vOUs', "aujOUrd'hui", '?']
>>> triples.splitlines()      # Découpage par lignes
['ou triples qui seuls acceptent', 'les sauts de ligne.']
```

Code *AP1*.9: Opérations sur les listes

```
>>> L = [1,3,2,8,42,53,33]# Une liste se définit entre crochets
>>> # Possibilité d'extraire des sous-listes, dernier élément exclu (cf range)
>>> L[1:3],L[:5],L[3:],L[1:6:2]
([3, 2], [1, 3, 2, 8, 42], [8, 42, 53, 33], [3, 8, 53])
>>> Lsave = L[:]           # Manière standard de vraiment *copier* une liste
>>> L2= L                  # Alors que là, L et L2 *pointent* vers le même objet
>>> L.append(155)          # Rajout d'un élément a la fin
>>> L2                     # ce qui modifie *aussi* L2 ! (en apparence)
[1, 3, 2, 8, 42, 53, 33, 155]
>>> L[0],L[5],L[-1],L[-3] # La numérotation commence à 0.
(1, 53, 155, 53)
>>> len(L)                 # len() donne le nombre d'éléments de la liste
8
>>> L.sort()               # L.sort() modifie la liste
>>> L                      # dont l'ordre est donc modifié
[1, 2, 3, 8, 33, 42, 53, 155]
>>> sorted(Lsave)          # alors que sorted() fait une copie de la liste
[1, 2, 3, 8, 33, 42, 53]
>>> Lsave                  # laissant la liste de départ inchangée
[1, 3, 2, 8, 42, 53, 33]
>>> L.pop(2), L.pop()      # pop() enleve un élément (par defaut le dernier)
(3, 155)
>>> L.remove(42)           # remove() supprime un objet (le premier trouvé)
>>> L                      # Voici ce qu'il reste
[1, 2, 8, 33, 53]
>>> L.insert(2,'Hello !') # Au contraire, insert() en insère un à la place voulue
>>> L
[1, 2, 'Hello !', 8, 33, 53]
>>> L.reverse()            # Et reverse() renverse la liste
>>> L                      # qui est modifiée !
[53, 33, 8, 'Hello !', 2, 1]
>>> L.index('Hello !')     # On peut aussi rechercher la place d'un élément connu
3
>>> L + Lsave              # Finalement, on peut concaténer deux listes en une 3e
[53, 33, 8, 'Hello !', 2, 1, 1, 3, 2, 8, 42, 53, 33]
>>> L                      # ce qui ne modifie pas les listes de départ
[53, 33, 8, 'Hello !', 2, 1]
>>> L.extend(Lsave)        # ou modifier L en utilisant extend()
>>> L
[53, 33, 8, 'Hello !', 2, 1, 1, 3, 2, 8, 42, 53, 33]
>>> [0,1]*10               # ou encore initialiser une liste de manière synthétique
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

Code *AP1*.10: Expression ou instruction ?

```
>>> x = 42**2
>>> abs(144 - x) * 3 + 42 * 5
5070
>>> if (x > 255):
...     print('x est grand !')
... else:
...     print('Essaie encore...')
...
x est grand !
>>> y = print('Oups !?!')
Oups !?!
>>> print(y)
None
```

Code *AP1*.11: Affectations

```
>>> x = 42      # Affectation à un littéral
>>> y = 2 - x**2# Affectation à une expression
>>> z = abs(y)  # Affectation au résultat d'une fonction
>>> x,y,z
(42, -1762, 1762)
```

Code *AP1*.12: Affectations multiples

```
>>> # Sans affectation multiple
>>> x = 1      # x vaut 1
>>> y = 2      # y vaut 2
>>> x,y        # Vérification
(1, 2)
>>> # Début de l'échange
>>> temp = x  # Création d'une variable temporaire pour stocker x
>>> x = y      # Nouvelle valeur de x (2)
>>> y = temp  # y récupère l'ancienne valeur de x (1)
>>> x,y        # Vérification
(2, 1)

>>> # Avec affectations multiples
>>> x,y = 1,2 # Affectations
>>> x,y        # Vérification
(1, 2)
>>> x,y = y,x # Échange
>>> x,y        # Vérification
(2, 1)
```

Code *AP1*.13: Attention aux affectations d'objets mutables comme les listes !

```
>>> x = [1,2,3,4]       # x pointe sur une liste
>>> y = [1,2,3,4]       # y pointe sur une *autre* liste (de mêmes valeurs)
>>> z = y               # z et y pointent sur la *même* liste
>>> t = y[:]            # Alors que t pointe sur une *copie* de y (donc différente)
>>> for L in [x,y,z,t]:# Boucle de vérification des
...     id(L)           # identifiants (emplacements mémoire)
...
4428876472
4428834144
4428834144
4428900472
>>> z[2] = 42           # Modification uniquement (en apparence) de z
>>> x,y,z,t             # Qui se reflète aussi sur y ! (mais ni sur x ni sur t)
([1, 2, 3, 4], [1, 2, 42, 4], [1, 2, 42, 4], [1, 2, 3, 4])
```

Code *AP1*.14: Raccourcis d'affectations: que vaut i tout à la fin ?

```
>>> i  = 1
>>> i += 10   # équivalent à i = i + 10
>>> i -= 7    # équivalent à i = i -  7
>>> i**= 3    # équivalent à i = i**3
>>> i %= 4    # équivalent à i = i%4

>>> i
0
```

Code *AP1*.15: Condition simple

```
if condition1:         # Notez le début du bloc avec le ":"
    instruction1            # Ce bloc (instruction1 à 3)
    instruction2            # n'est exécuté que si condition1
    instruction3            # est évaluée à True
main_instruction1       # Celui-ci en revanche est exécuté
main_instruction2       # quoi qu'il arrive
```

Code *AP1*.16: Condition avec alternative

```
if condition1:          # À nouveau, début du bloc après le ":"
    instruction1              # Ce bloc (instruction1 à 3)
    instruction2              # n'est exécuté que si condition1
    instruction3              # est évaluée à True
else:                   # Le 'else' aussi a droit a son ":"
    instruction4              # Ce bloc en revanche (instruction4 à 6)
    instruction5              # ne sera exécuté que si condition1
    instruction6              # a été évaluée à False
main_instruction1       # Retour au programme principal,
main_instruction2       # exécuté quoi qu'il arrive
```

Code *AP1*.17: Condition avec branchements multiples

```
if condition1:          # À nouveau, début du bloc après le ":"
    instruction1              # Ce bloc (instruction1 à 3)
    instruction2              # n'est exécuté que si condition1
    instruction3              # est évaluée à True
elif condition2:        # Le 'elif' aussi a droit a son ":"
    instruction4              # Ce bloc en revanche (instruction4 à 7)
    instruction5              # ne sera exécuté que
    instruction6              #  si  condition1 a été évaluée à False
    instruction7              # *et* condition2 a été évaluée à True
elif condition3:        # Le 'elif' aussi a droit a son ":"
    instruction8              # Ce bloc, quant à lui, (instruction8 à 12)
    instruction9              # ne sera exécuté que
    instruction10             #  si  condition1 a été évaluée à False
    instruction11             # *et* condition2 a été évaluée à False
    instruction12             # *et* condition3 a été évaluée à True
else:                   # Le 'else' aussi a droit a son ":"
    instruction13             # Ce bloc, finalement, (instruction13 à 16)
    instruction14             # ne sera exécuté que si *aucune* des conditions
    instruction15             # préalables n'a été évaluée à True, ce qui veut dire
    instruction16             # qu'elles ont toutes été évaluées à False
main_instruction1       # Retour au programme principal,
main_instruction2       # exécuté quoi qu'il arrive
```

Code *AP1*.18: Instructions conditionnelles imbriquées

```
# Version avec conditions imbriquées
if Aurore_est_sage:
    if Eleonore_est_sage: dorment_paisiblement_dans_leur_chambre()
    else: Eleonore_est_separee()
else: Aurore_est_separee()

# Version sans imbrication
if Aurore_est_sage and Eleonore_est_sage: dorment_paisiblement_dans_leur_chambre()
elif Aurore_est_sage and not Eleonore_est_sage: Eleonore_est_separee()
else: Aurore_est_separee()
```

Code *AP1*.19: Mémory de Traque à la chaussette dans la JungleSpeed: quel est la bonne manière de coder ?

```
if unique_aventurier:   print("L'aventurier commence !")
if unique_plus_jeune:   print("Le plus jeune commence !")
if unique_original  :   print("Les chaussettes originales commencent !")
else:                   print("Vous n'avez qu'à tirer au sort...")

if unique_aventurier:   print("L'aventurier commence !")
elif unique_plus_jeune: print("Le plus jeune commence !")
elif unique_original  : print("Les chaussettes originales commencent !")
else:                   print("Vous n'avez qu'à tirer au sort...")
```

Code *AP1*.20: Intégration temporelle avec boucle `while`

```
>>> t = 0               # Temps initial
>>> dt= 0.1             # Incrément de temps
>>> while t < 0.3:      # La boucle continue tant que t<0.3
...     t += dt
...
>>> print('Temps de sortie de boucle:',t)
Temps de sortie de boucle: 0.30000000000000004

>>> # Attention aux problèmes de précision des flottants...
>>> t = 0               # Temps initial
>>> dt= 0.1             # Incrément de temps
>>> while t != 0.3 and t < 2:
...     t += dt
...
>>> print('Temps de sortie de boucle:',t)
Temps de sortie de boucle: 2.0000000000000004
```

Code *AP1*.21: Compteur avec boucle `while`

```
>>> i,L = 0,[]      # Initialisation du compteur et de la liste
>>> while i < 10:
...     L.append(i**2)
...     i += 1
...
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Code *AP1*.22: Boucle `while` à nombre indéterminé d'itération à l'avance

```
>>> import random             # Module 'random' pour le mélange
>>> i = 0                     # Initialisation du compteur
>>> L = list(range(20))       # Initialisation de la liste
>>> random.shuffle(L)         # Mélange de la liste
>>> inferieur_a_18 = True     # Initialisation de la condition
>>> while inferieur_a_18:
...     if L[i] >= 18:        # Condition d'arrêt choisie
...         inferieur_a_18 = False
...         resultat = i      # On stocke le résultat
...     i += 1
...
>>> resultat,i                # À noter que i a été incrémenté une fois de plus
(11, 12)
>>> L                         # Vérification sur L
[5, 0, 2, 1, 7, 17, 4, 10, 12, 16, 11, 18, 3, 14, 8, 13, 19, 15, 9, 6]
>>> # Le même résultat avec l'instruction break pour sortir de la boucle
>>> i = 0                     # Réinitialisation
>>> while True:
...     if L[i] >= 18: break # break impose une sortie directe de la boucle
...     i += 1                # Instruction exécutée si break ne l'a pas été
...
>>> i                         # i stocke donc la bonne valeur
11
```

Code *AP1*.23: Boucles `for`, introduction: création du carré d'une liste

```
>>> L = [1,5,2,9,42]             # Liste de départ
>>> newL = []                    # Initialisation de la nouvelle liste
>>> for v in L:                  # Boucle sur les valeurs v de la liste
...     newL.append(v**2)        # Ajout du carré v**2 à newL
...
>>> newLbis = [v**2 for v in L] # Néanmoins, Python offre une syntaxe
>>> newL,newLbis                 # plus intuitive pour ce cas précis
([1, 25, 4, 81, 1764], [1, 25, 4, 81, 1764])
```

Code *AP1*.24: Boucles `for`, utilisation de `range`

```
>>> list(range(10))      # 10 éléments, de 0 à 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10))    # On commence à 2, on incrémente de 1 tant que <10
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10,3)) # On commence à 2, on incrémente de 3 tant que <10
[2, 5, 8]
```

Code *AP1*.25: Exemple d'implémentation de `range` à l'aide d'une boucle `while`

```
>>> def range(stop,start=0,step=1):
...     '''Implémentation "manuelle" de la fonction range de Python.
...     l'ordre des paramètres n'est pas respecté car Python se permet
...     des chose qu'il ne permet pas à ses utilisateurs...'''
...     i,L = start,[]
...     while i < stop: L.append(i) ; i += step
...     return L
...
>>> range(stop=10),range(start=2,stop=10),range(start=2,stop=10,step=3)
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [2, 3, 4, 5, 6, 7, 8, 9], [2, 5, 8])
```

Code *AP1*.26: Boucles `for` utilisant `range`

```
>>> L2,L3,L4 = [],[],[]         # Initialisation des listes
>>> for i in range(10):         # Fabrication de la liste des carrés
...     L2.append(i**2)         # par rajouts successifs
...
>>> L2                          # Vérification
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> for i in range(len(L2)):    # Fabrication de la liste de i^2*(i-2)
...     L3.append(L2[i]*(i-2)) # en utilisant la liste précédente
...
>>> L3                          # Vérification
[0, -1, 0, 9, 32, 75, 144, 245, 384, 567]

>>> # Un exemple utilisant break (déconseillé car moins lisible)
>>> for i in range(10):
...     if i**4 > 1e3: break    # Condition d'arrêt avec break
...     L4.append(i)            # (pour éviter de le calculer "à la main" avant)
...
>>> L4,5**4,6**4                # Vérification
([0, 1, 2, 3, 4, 5], 625, 1296)
```

Code *AP1*.27: Exemple de fonction inutile

```
>>> def fonction_inutile(a,b):
...     '''Une fonction totalement inutile car elle ne fait rien...
...     Néanmoins, pour l'appeller, vous devez passer 2 arguments
...     (que vous pouvez choisir de manière totalement aléatoire...)'''
...     pass
...
>>> fonction_inutile(0,1)  # On vérifie que rien ne se passe
>>> help(fonction_inutile) # Quelqu'un a demandé de l'aide ?
Help on function fonction_inutile:

fonction_inutile(a, b)
    Une fonction totalement inutile car elle ne fait rien...
    Néanmoins, pour l'appeller, vous devez passer 2 arguments
    (que vous pouvez choisir de manière totalement aléatoire...)
```

Code *AP1*.28: Exemple de fonction calculant π

```
# La fonction proprement dite
def calcul_de_pi(n=10):
    '''Calcul de pi en utilisant la formule de sommation des impairs alternés.
    Par défaut, on va jusqu'à 10 termes sommés, mais on peut comparer
    la précision en rajoutant quelques termes deci-delà.'''
    somme = 0             # Initialisation de la somme
    for k in range(n):    # Boucle sur n termes sommés
        somme += (-1)**k / (2*k+1.0)  # Rajout d'un terme supplémentaire
    return 4*somme        # Renvoi de la valeur finale

# Vérification pour différentes valeurs de n
def affichage_calculs_de_pi(puissance_max=6):
    '''Procédure qui renvoie sous forme de chaîne de caractère
    les résultats de l'application de la fonction 'calcul_de_pi'
    pour diverses valeurs de n sur une échelle logarithmique'''
    stot = ''
    for i in range(puissance_max):  # Les puissances de 10
        for j in range(5,11,5):     # Quelques valeurs par puissance
            n = j * 10**i
            # On coupe en plusieurs chaînes pour raisons de lisibilité
            s1= 'Pour n=' + str(n).rjust(puissance_max+1) + ', '
            s2= 'le calcul donne pi=' +  str(calcul_de_pi(n))
            stot+= s1 + s2 + "\n"
    return stot                     # et on renvoie la chaîne finale

print(affichage_calculs_de_pi())
```

Code *AP1*.29: Résultat des calculs de π

```
Pour n=     5, le calcul donne pi=3.33968253968
Pour n=    10, le calcul donne pi=3.04183961893
Pour n=    50, le calcul donne pi=3.12159465259
Pour n=   100, le calcul donne pi=3.13159290356
Pour n=   500, le calcul donne pi=3.13959265559
Pour n=  1000, le calcul donne pi=3.14059265384
Pour n=  5000, le calcul donne pi=3.14139265359
Pour n= 10000, le calcul donne pi=3.14149265359
Pour n= 50000, le calcul donne pi=3.14157265359
Pour n=100000, le calcul donne pi=3.14158265359
Pour n=500000, le calcul donne pi=3.14159065359
Pour n=1000000, le calcul donne pi=3.14159165359
```

Code *AP1*.30: Arguments et résultats d'une fonction

```python
# Fonction à trois paramètres, tous obligatoires
# Renvoie un triplet de valeurs
def permutation_circulaire(a,b,c):
    '''Renvoie la permutation circulaire des trois valeurs
    (paramètres obligatoires) données en entrée
    (aux noms de variables peu évocateurs...)'''
    return b,c,a

from math import * # Pour importer cos, sqrt et pi (et quelques autres...)

# Fonction à deux paramètres obligatoire et un optionnel
# Renvoie une unique valeur.
def al_kashi(a,b,theta=pi/2):
    '''Renvoie la valeur du 3e côté (c) d'un triangle, connaissant
    les deux autres côtés (a et b) ainsi que l'angle theta opposé.
    L'angle theta est supposé être donné en radians. Par défaut,
    on suppose que le côté c est l'hypothénuse d'un triangle rectangle.'''
    return sqrt(a**2 + b**2 - 2*a*b * cos(theta))

# Testons les deux fonctions précédentes
a,b,c = 1,2,3
a,b,c = permutation_circulaire(a,b,c)
print(a,b,c)

print('c = ',al_kashi(3,4))       # Triangle pythagoricien (3,4,5)
print('c = ',al_kashi(4,4,pi/3)) # Triangle équilatéral
print('c = ',al_kashi(4,4,pi/6)) # Triangle isocèle
```

Code *AP1*.31: Résultat des tests précédents

```
(2, 3, 1)
c = 5.0
c = 4.0
c = 2.07055236082
```

Code *AP1*.32: Fonctions permettant l'écriture et la lecture d'un fichier de données

```
# Fonction d'écriture dans un fichier.
# Ne renvoie rien, elle se contente  d'effectuer son boulot)
def ecriture(x,y,file='data.txt'):
    '''Avec deux listes de flottants de même taille en entrée, écrit les
    valeurs dans le fichier 'data.txt' (par défaut) à raison d'un couple de
    valeurs par ligne'''
    # Si x et y n'ont pas la même taille, pas la peine de continuer
    if len(x) != len(y):
        raise Exception("x et y n'ont pas le même nombre d'éléments...")
    f = open(file,'w')      # Ouverture du fichier en écriture (flag 'w': "write")
    for i in range(len(x)):# Boucle sur les différents éléments
        # On écrit les deux valeurs séparées par une tabulation
        f.write('{0}\t{1}'.format(x[i],y[i]))
    # Fermeture du fichier (important => l'écrire tout de suite après
    # l'ouverture pour ne surtout pas oublier !)
    f.close()

# La fonction en lecture du même fichier pour réaliser l'action inverse
# Elle renvoie les deux tableaux x et y
def lecture(file='data.txt'):
    '''Lecture du fichier (par défaut 'data.txt') qui est supposé contenir
    des couples de flottants à raison d'un par ligne (on ne va pas faire de
    gestion d'erreur sur ce coup là: c'est mal, mais c'est plus facile à
    écrire dans un premier temps). La fonction renvoie les deux listes de
    valeurs de x et de y'''
    f = open(file,'r')      # Ouverture du fichier en lecture seule ("read")
    x,y = [],[]             # Initialisation des sorties
    # le descripteur de fichier f est directement "itérable" sur les lignes
    # du fichier
    for line in f:
        # split() sépare par défaut selon les espaces et fait automatiquement
        # un strip() de la chaîne de caractères (suppression des espaces en
        # début et fin de ligne si ils sont présents)
        xi,yi = line.split()
        x.append(xi)        # Rajout de xi à la liste x
        y.append(yi)        # Rajout de yi à la liste y
    f.close()               # Ne *pas* oublier la fermeture
    return x,y              # Renvoi des listes x et y complétées.
```

Code *AP1*.33: Fonction plus évoluée pour physicien

```
# Fonction avec modification (apparente) d'un paramètre, mais en fait un
# flottant ou un entier n'est pas mutable (encore heureux !)
# Renvoie une unique valeur
def pression_gaz_parfait(V,n,T,celsius=False):
    '''Renvoie la pression d'un gaz parfait (en Pa) connaissant :
    * son volume V (en m^3);
    * son nombre de mol n;
    * et sa température T.
    Par défaut, la température doit être donnée en Kelvin,
    mais l'argument optionnel permet de la spécifier en degrés Celsius.'''
    R = 8.314                    # Définition de la constante des GP
    if celsius: T = T + 273.15   # Passage en Kelvin si demandé
    return n*R*T/V

# Quelques tests sur des valeurs "connues"
# Pour les CNTP (1atm, 0°C), on a Vm = 22.4 L
print('P_CNTP = ',pression_gaz_parfait(22.4e-3,1,273.15))
# Pour les CUTP (1atm, 20°C), on a Vm = 24 L
print('P_CUTP = ',pression_gaz_parfait(24e-3,1,20,True))
# NB: ne pas oublier la conversion des litres en m^3...

# CNTP = Conditions Normales de Température et de Pression
# CUTP = Conditions Usuelles de Température et de Pression
# 1atm = 101325 Pa
```

Code *AP1*.34: Résultat des tests précédents

```
P_CNTP = 101382.549107 Pa
P_CUTP = 101552.045833 Pa
```

Code *AP1*.35: Fonction plus évoluée pour mathématicien

```
from math import sqrt

# Une fonction avec deux 'return' selon qu'il existe ou non des solutions
# réelles.
# Renvoie, selon les cas, rien du tout (='None') ou un doublet de valeurs
def trinome(a,b,c,complexes = False):
    '''Fonction permettant de trouver les racines d'un trinôme du type ax^2+bx+c.
    Si elles existent, la fonction renvoie un doublet des deux racines qu'elle
    imprime de surcroît sur l'écran de l'utilisateur. Sinon elle renvoie
    'None' et affiche qu'il n'y a aucune racine réelle.
    On peut néanmoins imposer des solutions complexes avec l'argument optionnel'''
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        if complexes:    # Cas à racines complexes
            rac_disc = 1j * sqrt(-discriminant)
        else:            # Cas sans racines
            print("Ce trinôme n'admet pas de racines réelles")
            return None  # On sort directement de la fonction
    else:
        rac_disc = sqrt(discriminant)
    x1 = (-b - rac_disc)/(2.0*a)
    x2 = (-b + rac_disc)/(2.0*a)
    print('Les racines sont ' + str(x1) + ' et ' + str(x2))
    return x1,x2

# Tests sur des trinomes particuliers
print(trinome(1,-2,1))       # Racine double 1
print(trinome(1,1,1))        # j^2 + j + 1 = 0 pour le mathématicien
print(trinome(1,1,1,True))   # il faut activer les complexes pour que cela marche
print(trinome(1,0,1,True))   # i et -i
print(trinome(1,0,-2))       # sqrt(2) et -sqrt(2)
```

Code *AP1*.36: Résultat des tests précédents

```
Les racines sont 1.0 et 1.0
(1.0, 1.0)
Ce trinôme n'admet pas de racines réelles
None
Les racines sont (-0.5-0.866025403784j) et (-0.5+0.866025403784j)
((-0.5-0.8660254037844386j), (-0.5+0.8660254037844386j))
Les racines sont -1j et 1j
(-1j, 1j)
Les racines sont -1.41421356237 et 1.41421356237
(-1.4142135623730951, 1.4142135623730951)
```