Chapitre 1. Premiers pas en python

Voir l'annexe C pour une description de l'environnement Python Tutor.

1.1 Affectation et expressions

python permet tout d'abord de faire des calculs. On peut évaluer des expressions (arithmétiques ou booléennes, par exemple). Il faut pour cela respecter la syntaxe du langage. On peut sauvegarder des valeurs dans des variables. Chaque variable a un nom. Par exemple, pour sauvegarder la valeur 12×5 dans une variable qui s'appelle `x`, on tape l'instruction

```
x = 12 * 5
```

On peut ensuite réutiliser `x`, et sauvegarder la valeur de $\texttt{x}^2$ dans la variable de nom `y` en tapant l'instruction `y = x * x`. Contrairement à sa signification mathématique, le symbole `=` signifie « *calculer la valeur à droite du signe* `=` *puis mémoriser le résultat dans la variable dont le nom est à gauche du signe* `=` », il y a donc deux étapes bien distinctes : calculer d'abord, et stocker le résultat ensuite.

Les instructions sont exécutées une à une et dans l'ordre dans lequel elles sont écrites : elles sont exécutées en *séquence*. Le tableau suivant illustre l'évolution des valeurs des variables `x`, `y` et `z` lors de l'exécution séquentielle des quatre instructions suivantes :

```
x = 12 * 5
y = x * x
z = x + 1
x = y + z
```

	étape 1	*étape 2*	*étape 3*	*étape 4*
x	**60**	60	60	**3661**
y		**3600**	3600	3600
z			**61**	61

$\longrightarrow$ *temps*

On observe que dans chaque colonne de ce tableau, une seule case est en gras ; elle correspond à la variable affectée à l'étape correspondante (l'étape *1* correspond à l'exécution de la première instruction, etc . . .), les autres variables ne sont pas affectées. On notera également que l'affectation de la variable `x` à l'étape *4* n'a pas d'effet sur les variables `y` et `z`. La valeur finale d'une variable est la valeur dans la dernière colonne. La valeur finale de `x` est donc 3661, celle de `y` est 3600 et celle de `z` est 61.

Pour améliorer la lecture et faciliter l'écriture de ce style de tableau, il est intéressant d'écrire uniquement les valeurs des variables affectées, étape par étape. La valeur finale d'une variable est maintenant la valeur la plus à droite sur sa ligne, il s'agit toujours de la valeur calculée lors la dernière affectation de la variable.

	étape 1	*étape 2*	*étape 3*	*étape 4*
x	60			**3661**
y		**3600**		
z			**61**	

$\longrightarrow$ *temps*

Dans le tableau ci-dessus une seule valeur apparaît par colonne car dans le programme correspondant, comme dans tous ceux de ce chapitre, une affectation ne concerne qu'une seule variable à la fois. Dans les exercices qui suivent, nous vous recommandons fermement d'utiliser un tel tableau pour montrer l'évolution des valeurs des variables. On veillera à ne faire apparaître qu'une seule affectation par colonne pour bien mettre en valeur la chronologie des affectations.

Exercice 1.1.1 Que contiennent les variables `x`, `y`, `z` après les instructions suivantes ?

```
x = 6
y = x + 3
x = 3
z = 2 * y - 7
```

	étape 1	*étape 2*	*étape 3*	*étape 4*
x				
y				
z				

Exercice 1.1.2 L'instruction `i = i + 1` a-t-elle un sens ; si oui lequel ? Et `i + 1 = i` ?

Exercice 1.1.3 Que contiennent les variables `x`, `y`, `z` après les instructions suivantes ?

```
x = 6
y = 7
z = x
z = z + y
```

	étape 1	*étape 2*	*étape 3*	*étape 4*
x				
y				
z				

Exercice 1.1.4 Quel est le résultat de l'instruction `x = 2 * x` ? Si la valeur initiale de `x` est 1, donner les valeurs successives de `x` après une, deux, trois, etc. exécutions de cette instruction.

	étape 1	*étape 2*	*étape 3*	*étape 4*	*étape 5*	*étape 6*	*étape 7*	...	*étape n*
x	1								

Exercice 1.1.5 Parmi les codes suivants, quels sont les programmes python qui s'exécutent sans causer d'erreur ? Indiquer les erreurs dans les codes erronés.

```
x = 1
x = y + 1
y = 2
```

```
y = 2
x = 1
x = y +- 2
```

```
y = 2
x = 1
x = y + 3
```

```
x = 1
y = 0
x + y = 1
```

```
y = 2
x = 1
x = y +/- 1
```

```
y = 2
x = 1
y = x -+ 1
```

Exercice 1.1.6 Écrire une suite d'instructions permettant d'échanger le contenu de deux variables `a` et `b`.

Divisions entières

En termes simples, la division entière est une division arrondie « par défaut », c'est-à-dire que l'on ne conserve pas les chiffres après la virgule dans le résultat (le quotient) : $17/5 = 3,4$ mais on ne conserve que 3. Il y a alors un reste : $17 - 5 * 3 = 2$

Plus formellement, le quotient entier q de deux entiers a et b positifs, et le reste r de la division sont définis par :

$$a = bq + r \qquad \text{avec} \quad 0 \leq r < b$$

Par exemple le quotient et le reste de la division de 17 par 5 sont 3 et 2 car $17 = 5 \times 3 + 2$.

En python le quotient entier de a par b est noté `a//b`, et le reste `a%b` (aussi appelé modulo).

Exercice 1.1.7 Taper les instructions suivantes et prenez le temps d'expliquer précisément l'évolution des variables pendant l'exécution pas à pas de ces instructions. Les parties à droite des dièses `#` ne sont que des commentaires pour vous, l'ordinateur ne les interprètera pas même si vous les tapez.

```
x = 11 * 34
x = 13.4 - 6
y = 13 / 4    # division avec virgule
y = 13 // 4   # division entiere, notez la difference
z = 13 % 2    # pourquoi cela vaut-il 1 ?
x = 14 % 10   # quel sens donner au reste d'une divison par 10 ?
y = 14 // 10
i = x + y
```

x															
y															
z															
i															

1.2 Fonctions

La plupart des fonctions servent à *calculer* un résultat, comme en mathématiques, et à le *retourner* (on dit aussi *renvoyer*). La syntaxe python pour la définition d'une fonction est la suivante :

```
def nom_fonction(liste, de, parametres):
  corps de la fonction
```

La définition d'une fonction, effectuée en général une seule fois, permet de définir les paramètres de la fonction et le corps de la fonction. Remarquez les deux points à la fin de la première ligne. Les instructions qui constituent le corps de la fonction doivent être indentées par rapport au mot clef `def`, c'est-à-dire décalées, pour indiquer à python qu'elles font partie de la définition de la fonction.

L'instruction `return` termine l'exécution de la fonction, elle peut être suivie d'une expression pour indiquer la valeur renvoyée par la fonction.

Dans l'exemple ci-dessous, la fonction `f` a un seul paramètre, nommé `x`, et le corps de la fonction `f` est constitué de deux instructions.

Exemple de définition et d'appels de fonction :

```
# définition de la fonction f
def f(x):
  a = x+1
  return a*a + x + 1

# appel de la fonction f avec l'argument 5
y = f(5)

# appel de la fonction f avec l'argument y + 1
z = f(y + 1)
```

Une fonction doit être appelée pour être exécutée et peut être appelée autant de fois que l'on veut. Un appel de fonction fournit les arguments de cet appel et il y a autant d'arguments qu'il y a de paramètres dans la définition de la fonction.

Comme pour l'instruction d'affectation, l'appel d'une fonction se fait en plusieurs étapes bien distinctes : les valeurs des arguments passés à la fonction sont d'abord calculées. La fonction est alors appelée avec le résultat de ces calculs. Le corps de la fonction est alors exécuté, les paramètres contenant alors les résultats des calculs des arguments. La fonction se termine au premier `return` exécuté qui désigne la valeur à retourner. L'exécution revient alors à l'endroit où l'on a effectué l'appel à la fonction, et c'est la valeur retournée par la fonction qui y est utilisée.

```
# definition de la fonction g contenant du code mort
def g(x):
  a = x+1
  return a*a + x + 1
  # code mort - erreur de programmation a eviter
  b = a + 1
```

La définition de fonction ci-dessus contient du code mort : les instructions qui suivent une instruction `return` ne sont jamais exécutées, c'est une erreur de programmation.

Exercice 1.2.1 Après avoir défini la fonction `f` comme ci-dessus, taper les instructions suivantes qui appellent cette fonction en lui passant différents arguments et prenez le temps d'expliquer en détail les résultats obtenus.

```
x = 0
y = f(2)
t = 4
y = f(t)              # on passe la valeur d'une variable
y = f(1) + f(2)       # on effectue deux appels
z = x+1
y = f(z)
y = f(x+1)            # on passe directement la valeur d'une expression
z = f(x-t)
t = f(t)              # on peut meme passer la variable qui servira a
                      # stocker le resultat
x = f(f(1))           # on peut combiner deux appels, le resultat de
                      # l'un est passe en parametre a l'autre
```

Exercice 1.2.2 Parmi les codes suivants, quels sont les programmes python qui ne comportent pas d'erreur ? Rayer les codes erronés.

```
Def fun(x):
  return x + 1

y = fun(3)
```

```
def fun(x):
return x + 1

y = fun(3)
```

```
def fun(x)
  return x + 1

y = fun(3)
```

```
def fun(y):
  return x + 1

y = fun(3)
```

```
def fun(x):
  return x + 1

y = fun(3)
```

```
def fun():
  return 2

y = fun(3)
```

Exercice 1.2.3

1. Soient `x` et `y` deux variables contenant chacune un nombre. Écrire l'expression python qui stocke dans une variable `m` le calcul de la moyenne des deux nombres `x` et `y`.

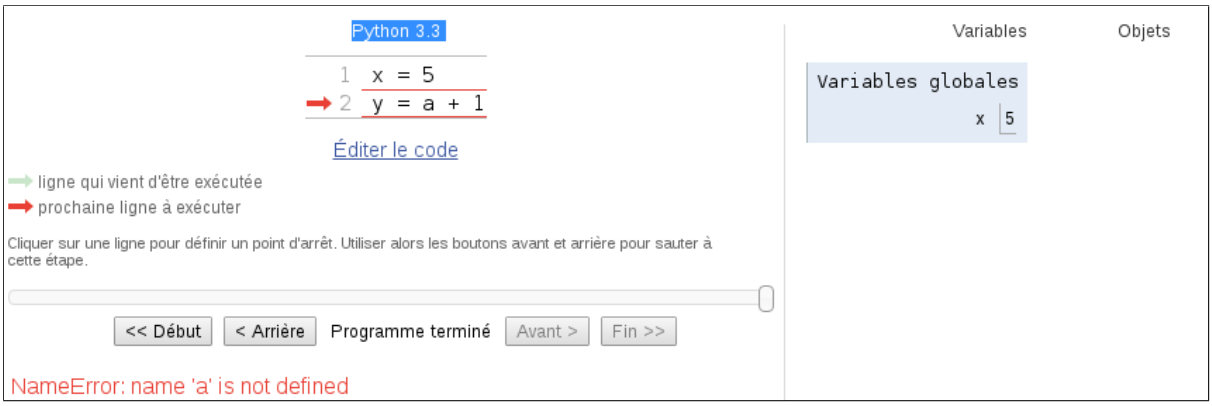
2. Écrire une fonction `moyenne(a,b)` qui retourne la moyenne des deux nombres `a` et `b`. Testez-la avec les arguments 42 et 23.

3. Écrire une fonction `moyennePonderee(a, coef_a, b, coef_b)` qui retourne la moyennne pondérée par le coefficient `coef_a` pour la note `a` et par le coefficient `coef_b` pour la note `b`. Testez-la en appelant `moyennePonderee(5,2,12,3)`.

4. Utilisez votre fonction `moyennePonderee(a, coef_a, b, coef_b)` pour écrire une autre version de la fonction `moyenne(a,b)` *(question 2.)*. Testez-la.

Compléments de programmation

Une erreur classique de programmation consiste à utiliser une variable qui n'est pas définie. C'est le cas de la variable `a` dans le code ci-dessous. Cette erreur est signalée par un message et provoque l'arrêt de l'exécution du programme.



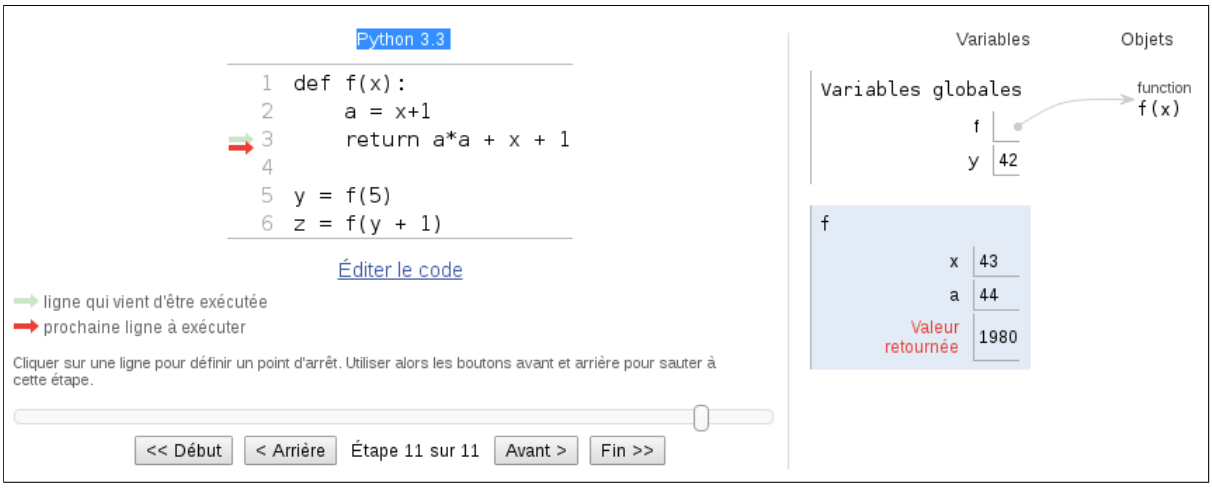
La figure ci-dessous illustre la distinction entre les variables globales et locales :

- les variables globales sont définies en dehors de toute fonction ; par exemple, la variable `y` de valeur 42.

- les variables locales à une fonction sont les paramètres de la fonction et les variables définies dans son corps ; par exemple, les variables `x` et `a` pour la fonction `f`. Ces variables n'existent que pour la durée de l'exécution de la fonction (lors d'un appel).

Il est possible d'utiliser (lire) une variable globale dans le corps d'une fonction. Pour des raisons de lisibilité du code, nous n'utiliserons pas cette possibilité : dans le corps des fonctions que nous écrirons, nous utiliserons uniquement les variables locales à cette fonction.

De plus, à notre niveau, il est préférable d'éviter d'avoir des variables globales et locales de même nom.



1.3 Conditionnelles

Expressions booléennes

Une expression booléenne est une expression qui n'a que deux valeurs possibles, `True` (vrai) ou `False` (faux) ; les tests `x == y` (égalité), `x != y` (inégalité), `x < y`, `x <= y`, `x >= y`, etc. sont des expressions booléennes. On peut combiner des expressions booléennes avec les opérateurs `and`, `or` et `not`.

Exercice 1.3.1 Taper les instructions suivantes et prenez le temps d'expliquer précisément l'évolution des variables pendant l'exécution pas à pas de ces instructions.

```
i = 9
j = 0
b = i < j     # b ne contient donc pas un entier, mais True ou False
b = i != 9
b = i == 9
bi = i % 2 == 0 or i % 3 == 0
bj = j % 2 == 0 or j % 3 == 0
b = not (bi and bj)
```

Exercice 1.3.2 Parmi les expressions suivantes, quelles sont celles qui valent `True` *si et seulement si*[1] les trois variables `a`, `b` et `c` ont des valeurs toutes différentes ? Rayer les expressions incorrectes.

- `a != b and b != c`
- `a != b and b != c and a != c`
- `a != b or b != c`
- `a != b or b != c or a != c`

Exercice 1.3.3 Écrire une expression qui vaut `True` si `x` appartient à $[0, 5[$ et `False` dans le cas contraire.

Écrire une expression qui vaut `True` si `x` n'appartient **pas** à $[0, 5[$ et `False` dans le cas contraire. Proposez-en deux formes différentes : l'une avec une négation, et l'autre sans.

1. *si et seulement si* signifie que l'un est vrai si l'autre est vrai, et que l'un est faux si l'autre est faux.

Exercice 1.3.4 Parmi les expressions suivantes, quelles sont celles qui valent `True` si l'entier n est pair et `False` dans le cas contraire ? Rayer les mauvaises réponses.

- `n == 0 % 2`
- `1 != 2 % n`
- `n // 2 == 0`
- `n % 2 != 1`
- `0 != n % 2`
- `n % 2 == 0`

Écrire une fonction `pair(n)` qui teste si `n` est pair ; la valeur calculée doit être *booléenne*, c'est-à-dire égale à `True` ou `False`. Testez-la.

Exécution conditionnelle

La syntaxe `if ... else ...` permet de tester des conditions pour exécuter ou non certaines instructions.

```
if condition_1 :
    code a executer si conditon_1 est vraie
elif condition_2 :
    code a executer si condition_2 est vraie
    et condition_1 est fausse
else :
    code a executer si aucune condition est vraie
```

La partie `else` est optionnelle et la partie `elif` peut apparaître autant de fois que nécessaire. Les exemples suivant détaillent l'utilisation decette structure :

```
def prixCinema(age):
  prix = 10
  if age < 18:
    prix = 6.70
  return prix
```

```
def prixCinema(age):
  if age < 18:
    prix = 6.70
  else:
    prix = 10
  return prix
```

Ne pas oublier les deux points à la fin des lignes `if, else`. Les instructions à exécuter dans chacun des cas doivent être *indentées* et rigoureusement alignées verticalement (ici il y a quatre blancs au début de chaque instruction indentée) — en fait l'utilisateur est aidé dans cette tâche par le logiciel de programmation, qui insère les blancs à sa place. La partie `else` n'est pas obligatoire : son absence indique simplement qu'il n'y a rien à faire dans ce cas.

La syntaxe `if ... else ...` peut être imbriquée, par exemple cela permet de distinguer trois cas : $age \leq 14$, $14 < age < 18$ et $age \geq 18$. :

```
def prixCinema(age):
  if age <= 14:
    prix = 5
  else:
    if age < 18:
      prix = 6.70
    else:
      prix = 10
  return prix
```

On peut utiliser la syntaxe `if ... elif ... else ...` pour l'écrire de manière plus simple. On peut répéter la partie `elif` autant de fois que voulu. Sur cet exemple :

```
def prixCinema(age):
  if age <= 14:
    prix = 5
  elif age < 18:
    prix = 6.70
  else:
    prix = 10
  return prix
```

Cette fonction peut être écrite encore plus simplement en éliminant la variable locale `prix` grâce à l'instruction `return` :

```
def prixCinema(age):
  if age <= 14:
    return 5
  elif age < 18:
    return 6.70
  else:
    return 10
```

Finalement, comme l'instruction `return` *termine* l'exécution de la fonction, on peut plus encore simplifier ce code :

```
def prixCinema(age):
  if age <= 14:
    return 5
  if age < 18:
    return 6.70
  return 10
```

Supposons que l'on exécute `prixCinema(17)`. La condition (`age <= 14`) du premier test n'étant pas satisfaite, le deuxième test (`if age < 18`) est alors évalué. Comme la condition (`age < 18`) est vraie, l'instruction `return 6.70` est exécutée et a pour effet de terminer l'évaluation de la fonction. Dans ce cas, l'instruction `return 10` n'est pas executée.

Exercice 1.3.5 Donner les valeurs des variables `x` et `y` après exécution de l'exemple suivant pour `x` valant 1, puis pour `x` valant 8.

```
if x % 2 == 0:
    y = x // 2
else:
    y = x + 1
x = x + 1
```

Même question pour chacun des codes suivants :

```
if x % 2 == 0:
    y = x // 2
    y = x + 1
x = x + 1
```

```
if x % 2 == 0:
    y = x // 2
else:
    y = x + 1
    x = x + 1
```

```
if x % 2 == 0:
y = x // 2
else:
    y = x + 1
x = x + 1
```

Exercice 1.3.6 Écrire une fonction `compare(a,b)` qui retourne -1 si $a < b$, 0 si $a = b$, et 1 si $a > b$. Testez-la. Donner plusieurs versions de cette fonction en vous inspirant de celles de la fonction `prixCinema`.

Exercice 1.3.7 Écrire une fonction `max2(x,y)` qui calcule et retourne la plus grande valeur parmi deux nombres x et y. Attention : bien nommer cette fonction `max2`, et non `max`, car la fonction `max` est prédéfinie en python.

1.4 Listes et boucles `for`

python permet de manipuler les listes d'éléments, par exemple `[2,8,5]` est une liste d'entiers. La fonction `len` (abréviation de *length*) retourne la *longueur* d'une liste (on dit aussi sa *taille*), par exemple `len([2,8,5])` vaut 3.

La boucle `for` permet de *parcourir* les éléments d'une liste ; en voici la syntaxe :

```
for variable in liste :
    corps de la boucle
```

Remarquez bien les deux points à la fin de la ligne et le fait que les instructions appartenant au corps de la boucle doivent être indentées (décalées) par rapport au mot clef `for`. L'évaluation par python d'une boucle `for` correspond à l'algorithme suivant :

1. calculer la valeur de la liste et la mémoriser ;
2. sortir de la boucle si tous les éléments de la liste ont été traités ;
3. affecter la valeur du premier élément non traité à la variable de boucle ;
4. exécuter le corps de la boucle ;
5. recommencer le traitement à partir de l'étape 2.

Exercice 1.4.1 Dans l'environnement Python Tutor, entrer l'instruction suivante et analyser ce qui est affiché :

```
for i in [2, 8, 5]:
    print(i, i * i)
```

Même question pour :

```
u = [2, 8, 5]
for i in u:
    print(i, i * i)
```

La variable `i` est-elle définie après exécution de ces instructions ? Si oui, quelle est sa valeur ?

Exercice 1.4.2 On considère la définition suivante :

```
def sommeListe(L):
    s = 0
    for i in L:
        s = s + i
    return s
```

Que retournent les appels `sommeListe([1,3,13])`, `sommeListe([1])` et `sommeListe([])` ? Pour vous aider, remplissez le tableau ci-dessous.

	étape 1	*étape 2*	*étape 3*	*étape 4*	*étape 5*	*étape 6*	*étape 7*	*étape 8*	*étape 9*
L	[1,3,13]								
i	-								
s	0								

L'instruction `return` termine l'exécution de la fonction qui la contient. Que calcule la fonction `somme` si par malheur on indente trop la dernière ligne, comme ci-dessous ?

```
def sommeListe(L):
    s = 0
    for i in L:
        s = s + i
        return s    # gros bug !
```

Exercice 1.4.3 Écart-type

1. En vous inspirant de la fonction `sommeListe` de l'exercice 1.4.2, écrivez une fonction `sommeCarresListe(L)` calculant la somme des carrés des éléments de la liste `L`.

2. L'écart-type d'une liste de nombres L permet d'estimer dans quelle mesure les éléments de L s'éloignent de la moyenne des éléments de L. Par exemple l'écart-type de la liste `[8, 8, 8, 12, 12, 12]` est de 2 (puisque tous les éléments sont à distance 2 de la moyenne 10).

   On peut le calculer en utilisant la somme des carrés des éléments de L et la somme des éléments de L (en notant n le nombre d'éléments de L) :

$$EcartType(L) = \sqrt{\frac{\sum_i {L_i}^2}{n} - \left(\frac{\sum_i L_i}{n}\right)^2}$$

   Pour calculer une racine carrée, il faut ajouter `from math import *` au début du fichier pour avoir accès à la fonction `sqrt` . En utilisant en plus les fonctions `sommeListe` et `sommeCarresListe` écrites précédemment, écrivez une fonction `ecartTypeListe(L)` qui calcule l'écart-type de la liste `L`.

Exercice 1.4.4 Écrire les fonctions suivantes prenant en paramètre une liste de nombres `L`, et **testez** ces fonctions :

- une fonction `moyenneListe(L)` qui calcule et retourne la moyenne de ces nombres,
- une fonction `maximumListe(L)` qui calcule et retourne le maximum de ces nombres (supposés positifs).
- une fonction `nbPairsListe(L)` qui compte et retourne combien de ces nombres sont pairs.

Vérification d'une propriété booléene sur une liste

Dans les exercices 1.4.3 et 1.4.4 nous avons effectué des mesures sur des listes, mesures dont l'évaluation nécessite la prise en compte de *tous les éléments de la liste* et, ce, *quel que soit la liste* considérée. Il existe cependant des propriétés qui peuvent être calculées sans nécessairement prendre en compte tous les éléments de la liste considérée. Par exemple, si l'on cherche à évaluer une propriété *"un des éléments de la liste est pair"*, alors on peut arrêter l'évaluation dès qu'un élément pair est rencontré : le résultat est alors déterminé et ne changera pas quelques soient la valeur des autres éléments. Dans le cadre de cet enseignement nous considérons que c'est une erreur de programmation de continuer un calcul alors que le résultat est déterminé. Dans le même registre, le code de la fonction suivante est aussi élégant qu'inefficace :

```
def existePairListeInefficace(L):          # gaspille de l'energie
    return nbPairsListe(L) != 0
```

Exercice 1.4.5 On considère les fonctions `existePairListe(L)`, qui renvoie `True` si au moins un des nombres de la liste est pair et `False` sinon, et `tousPairsListe(L)` qui renvoie `True` si tous les nombres de la liste sont pairs, et `False` sinon.

Écrire ces deux fonctions en faisant en sorte qu'elles retournent leur résultat dès que celui-ci est déterminé. Testez ces fonctions.

Une comparaison sur le nombre de tests réalisés par les fonctions `existePairListe(L)` et `existePairListeInefficace(L)` est présentée en Section 1.6.

1.5 Utilisation de range dans des boucles for

La fonction `range(debut, fin, pas)` permet de définir une suite arithmétique finie d'entiers de raison `pas` commençant par l'entier `debut` et bornée par `fin` (qui ne fait pas partie de la suite). Les syntaxes possibles de `range` sont :

```
range(debut, fin, pas)
range(debut, fin)       # pas vaut 1 par defaut
range(fin)              # debut vaut 0 par defaut
```

L'argument `pas` est donc optionnel, et vaut 1 par défaut. L'argument `debut` est également optionnel, et vaut 0 par défaut. La suite se termine **juste avant** d'atteindre `fin`.

Exercice 1.5.1 Entrer les instructions suivantes et analyser les réponses de python :

```
for j in range(10):
    print(j, j * j)
```

```
for k in range(3, 8):
    print(k, 2 * k + 1)
```

```
for k in range(3, 9, 2):
    print(k)
```

Exercice 1.5.2 Dans l'environnement Python Tutor, tester chacun des groupes d'instructions suivantes et analyser les résultats comme précédemment :