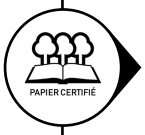
Internet des objets connectés

Cours, exercices et cas pratiques

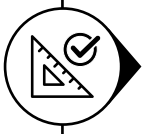
Thierry Alhalel
Rémi Boulle
Adrien van den Bossche

DUNOD

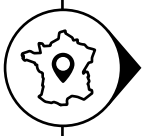
NOUS NOUS ENGAGEONS EN FAVEUR DE L'ENVIRONNEMENT :



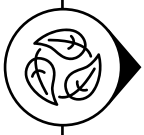
Nos livres sont imprimés sur des papiers certifiés pour réduire notre impact sur l'environnement.



Le format de nos ouvrages est pensé afin d'optimiser l'utilisation du papier.



Depuis plus de 30 ans, nous imprimons 70 % de nos livres en France et 25 % en Europe et nous mettons tout en œuvre pour augmenter cet engagement auprès des imprimeurs français.



Nous limitons l'utilisation du plastique sur nos ouvrages (film sur les couvertures et les livres).

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-100-85889-7

Sommaire

1

Notions sur le langage Python

Python est un langage de programmation interprété très utilisé dans le domaine de l'Internet des Objets. Il permet de traiter facilement de grands volumes de données venant de capteurs par exemple, de les représenter graphiquement, de les stocker en bases de données, etc. L'objectif de ce chapitre est de rappeler quelques notions de base concernant ce langage mais aussi de découvrir les principales libraires que vous allez utiliser dans vos projets IoT (Internet of Things, Internet des objets en anglais).

Sommaire

Il est entendu qu'il ne s'agit nullement de décrire ici le langage Python de manière exhaustive, mais bien de faire un résumé des notions nécessaires à l'objectif du chapitre.

Comme vous avez déjà étudié les rudiments du langage Python, vous savez que c'est un langage interprété (non compilé), orienté objet et sous licence libre. Ce langage est utilisé sous tous les principaux systèmes d'exploitation : Windows, GNU/Linux, Unix, Android et macOS.

Dans cet ouvrage, nous utiliserons la version Python 3. Il existe plusieurs façons d'écrire et de tester du code en Python. Vous pouvez bien sûr conser-

ver celle que vous avez apprise lors de vos études, mais sachez que les codes présentés ici sont testés :

- en environnement Linux avec l'éditeur Scite qui permet de développer et de tester du code Python de façon minimaliste ;
- en environnement Windows 7/10 ou Linux avec le logiciel Spyder 3 ;
- en environnement Anaconda, qui est une distribution libre open source de Python et R, installable sur les OS Windows et Linux, et permettant de traiter des données en quantité. Anaconda se présente comme un navigateur donnant accès en particulier à Spyder, Jupyter Notebook et RStudio.

L'une des principales forces de Python est l'existence de nombreuses librairies qui donnent accès à de puissantes possibilités graphiques et de calculs numériques.

Dans cet ouvrage, nous vous présenterons des programmes complets et fonctionnels soit dans les rappels de cours (la première ligne portera l'indication code *a.b*) soit dans les exercices (la première ligne comportera alors la mention exercice *a.b*). L'entier *a* renvoie au numéro de chapitre (ici le premier) et le second entier, *b*, indique le numéro de code au sein du chapitre. Vous pourrez utiliser cette nomenclature pour retrouver le code à télécharger sur le site de Dunod.

1.1 Les structures de base en Python

Nous ne ferons pas ici un cours de programmation, mais nous insisterons sur les notions essentielles du langage Python et de ses librairies.

1.1.1 Rappels de syntaxe

Le langage **Python** est naturellement **indenté** pour les blocs de code : boucles, sous-routines, tests conditionnels... vous devez absolument y faire attention. Voici un exemple de code fonctionnel avec indentation (tout ce qui est dans l'indentation fait partie du bloc d'instructions de la boucle) :

```
1  # code 1.1
2  # appel librairies
3  import random
4  import math
5  # initialisation des constantes et listes
6  N=1000;
7  DEDANS=0;
8  xdedans=[];
9  ydedans=[];
10 xdehors=[];
11 ydehors=[];
12 # boucle principale
```

```
for i in range(N):
    a, b = random.uniform(-1.,1.), random.uniform(-1.,1.)
    longueur = math.sqrt(a**2 + b**2)
    if longueur < 1:
        xdedans.append(a)
        ydedans.append(b)
  DEDANS=DEDANS+1
    else:
        xdehors.append(a)
  ydehors.append(b)
print(DEDANS)
```

On note dans le code qui précède, les indentations successives de la boucle `for` et de la condition `if`, car la conditionnelle (indentée) fait partie intégrante de la boucle (indentée). Autrement dit, **l'augmentation de l'indentation** signe le démarrage d'un nouveau bloc d'instructions.

Le code se structure de la façon suivante :

1. l'importation des librairies (ici la librairie `random` qui contient les lois de probabilités (la loi uniforme pour nous) et la librairie `math` qui contient les fonctions mathématiques classiques (racine carrée (sqrt) pour nous)) ;
2. la déclaration des constantes et l'initialisation des tableaux (il y en a quatre, lesquels ?) ;
3. le cœur du programme, avec le calcul de l'entier `DEDANS` qui est forcément inférieur ou égal à `N` (pourquoi ?) ;
4. le code permet également de remplir (instruction `append`) les quatre listes avec les valeurs de a et b qui répondent à la condition imposée par le `if`,
5. l'affichage final du résultat `DEDANS` (remarquons la position de cet affichage par rapport à la boucle `for`).

Les **commentaires** (sur une ligne) sont réalisés à l 'aide du caractère dièse : #. Il est important d'en mettre dans un code de façon à le rendre clair et compréhensible.

Dans le langage Python les variables sont typées **dynamiquement**, c'est-à-dire que le type d'une variable peut changer en cours d'exécution, comme sur le code ci-dessous :

```
# a est entier
a= 1
print('a est entier : ',a)
# a devient flottant
a=a*1.0
print('a devient flottant :', a)
```

avec le résultat suivant (on passe d'un entier à un type flottant) :

```
('a est entier : ', 1)
('a devient flottant :', 1.0)
```

1.1.2 Structures conditionnelles

La **structure conditionnelle** `if` se définit en Python :

```
if condition1:
    instructions1
else:
    instructions2
```

le **else** n'est pas obligatoire. Noter l'indentation utilisée.

On réalise une **répétition conditionnelle** en Python avec la boucle `while`.

```
while condition:
    instructions
```

Tant que la condition est vraie, la boucle `while` continue, et si le programme est mal conçu on peut tomber dans le piège de la **boucle infinie**, le programme ne peut alors pas se terminer. Pour illustrer le principe de la boucle `while`, on peut mettre en place l'**algorithme d'Euclide**, qui permet de trouver le **PGCD** (plus grand diviseur commun) de deux entiers strictement positifs a et b ($a \geq b$) :

Algorithme d'Euclide

- On affecte à la variable entière x la valeur entière a
- On affecte à la variable entière y la valeur entière b
- On affecte au compteur entier i la valeur 0

Tant que y est différent de 0 : (boucle while)

1. On incrémente le compteur i de 1 unité
2. On calcule le reste r_i de la division entière de x par y
3. On affecte à la variable x la valeur de y
4. On affecte à la variable y la valeur de r_i

À la sortie de la boucle `while` au bout de n itérations, le `PGCD(a,b)` est la dernière valeur de x (ou de y ce qui revient au même).

Par exemple si $a = 455$ et $b = 84$, le PGCD de ces deux entiers vaut 7 : on va représenter les étapes successives de l'algorithme qui mènent au résultat à l'aide d'un tableau qui représente ce qui se passe dans la boucle `while` (attention, c'est avant la boucle que x prend la valeur 455 et y la valeur 84) :

itération i	variable x	variable y	reste r_i
1	84	35	35
2	35	14	14
3	14	7	7
4	7	0	0

Voici la conclusion du programme : le PGCD de 455 et 84 est donc de 7. Le code Python correspondant pour la détermination du PGCD de deux entiers est :

```
# code1.2
# Entiers a et b dont on cherche le PGCD
a = 306
b = 756
# Test avec assert : les nombres sont bien positifs et a > b
assert (a > 0) and (b > 0)
if a < b:
    a, b = b, a # Interversion si besoin : on impose que a >= b


a0, b0 = a, b # On garde une copie des valeurs originales
# On boucle (while) jusqu a ce que le reste soit nul
while b != 0:
    a, b = b, a % b
# On remplace a par b et b par le reste de la division euclidienne de a par b

print('Le PGCD de {} et {} vaut : {}'.format(a0, b0, a) # On affiche le
    resultat
```

En voici le résultat :

```
Le PGCD de 756 et 306 vaut : 18
```

On réalise une **répétition inconditionnelle** (de N itérations) en Python avec la boucle `for` :

```
for i in range(1,N+1):
    instructions
```

Ici N est un entier, et i le compteur. Il est important de noter que l'entier i va varier de la valeur initiale 1 jusqu'à N.

Prenons comme exemple d'application la **fonction factorielle** d'un entier naturel n :

On définit en mathématiques la fonction factorielle d'un entier $n \geq 0$ (notée $n!$) comme :

$$\text{factorielle(n)} = n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

avec par convention :

$$0! = 1! = 1$$

et par exemple :

$$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$$

et

$$7! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 = 5040$$

et

$$8! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 = 40320$$

Voici un exemple de code qui calcule la factorielle de l'entier N à l'aide d'une boucle `for` :

```
# calcul de la  factorielle  de N
N=5
fact = 1
for i in range(1, N+1):
    fact = fact * i
print('la factorielle de {} est : {}'.format(N, fact)
```

Il faut bien remarquer que la valeur maximale du compteur i sera N et non $N + 1$, c'est-à-dire que i varie de 1 à N. et le résultat de ce code est :

```
'la factorielle de 5 est : 120'
```

Si on remplace dans le code précédent l'affectation $N = 5$ par $N = 6$ on obtient alors la sortie :

```
'la factorielle de 6 est : 720'
```

1.1.3 Listes

Une **liste** en Python est une suite, placée entre crochets, de valeurs (éléments) séparées par des virgules. Les éléments d'une liste ne sont pas obligatoirement tous du même type. Malgré son nom, une liste est proche d'un tableau.

Création d'une liste

On peut créer une liste de deux façons :

— **Extension :**
On liste simplement les éléments comme dans, par exemple :

```
    liste = [1, 1.2, 'a', "aa", [1, 2, 3]]
```

— **Compréhension :**
Méthode concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. Par exemple :

```
liste1 = [i**2 for i in range(100)]
liste2 = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]
```

génère la liste composée des carrés des entiers compris entre 0 et 99. La deuxième instruction génère celle des nombres pairs de 0 à 255 (on filtre avec `x % 2 == 0`) écrits sous formes de chaînes de caractères et en hexadécimal (0x...). La clause `if` est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

Accès aux élements

On peut accéder aux éléments d'une liste en utilisant l'indice de l'élément désiré ainsi : `liste[indice]`.

```
carres = [1, 4, 9, 16, 25, 6]
carres[0] # renvoie l'élément d'index 0
carres[-1] # renvoie le dernier
carres[5] = 36
```

L'instruction `carres[5] = 36` modifie le contenu de la liste et remplace l'élément d'indice 5 par 36.

On utilise très souvent des listes de nombres, et il est utile de connaître le nombre d'objets dans la liste. On donne ci-dessous un code qui calcule les 10 premières valeurs de la factorielle et les range dans la liste `factorielle[]`.

On y utilise la méthode `append()` et la primitive `len()` qu'il faut connaître.

```
#code1.3
#liste : on calcule les 10 premieres factorielles
#et on les range dans une liste : factorielle
# initialisation
N=10
factorielle=[]
#calcul des valeurs et remplissage de la liste
for j in range(1,N+1):
    fact= 1
    for i in range(1, j+1):
        fact = fact * i
        factorielle.append(fact) # ajoute fact àla fin de la liste

#affichage de la longueur de la liste avec la méthode len()
print('la liste contient : {} élements'.format(len(factorielle)))

#affichage du contenu de la liste
print('les elements de la liste sont : {}.format(factorielle))
```

L'affichage est le suivant :

```
la liste contient : 10 elements
les elements de la liste sont : [1, 2, 6, 24, 120, 720, 5040, 40320,
    362880, 3628800]
```

Slicing de listes

En anglais, *slice* signifie tranche. Ici, nous allons apprendre à découper des structures de données séquentielles (listes, tuples mais aussi des chaînes de caractères).

```
liste[start:stop] # de l'élément d'index start jusqu'à celui d'index stop-1
a[start:]   # de start jusqu'à la fin de la liste
a[:stop]    # du début jusqu'à stop-1
```

```
4 a[:]          # une copie de la  liste
```

Parcours d'une de liste

Une liste est un **itérable** (comme les chaînes de caractères). Cela signifie que l'on peut itérer dessus, c'est-à-dire répéter la même action. Le code ci-dessous affiche les carrés des entiers entre 0 et 99 :

```
1 liste = [i**2 for i in range(100)] # création de la liste en compréhension
2 for element in liste: # On itère sur la liste
3     print("le carré de {} est {}".format(i,element))
```

L'instruction `for` itère donc sur les éléments d'une séquence (qui peut être aussi une chaîne de caractères), dans leur ordre d'apparition dans la séquence.

1.1.4 Dictionnaires

Un autre type de donnée très utile est le dictionnaire. Les dictionnaires correspondent aux « tableaux associatifs » dans d'autres langages. À la différence des listes, indexées par des nombres, ils sont indexés par des clés.

Un dictionnaire est en fait un ensemble de couples clé-valeur au sein duquel les clés doivent être uniques.

Les opérations classiques sur un dictionnaire consistent à stocker une valeur pour une clé donnée et à extraire la valeur correspondant à une clé. Il est également possible de supprimer une paire clé-valeur avec `del`.

Création d'un dictionnaire

Comme pour les listes, on peut créer un dictionnaire de deux façons :

— **En extension** : on liste simplement les éléments comme dans, par exemple :

```
1 liste = [1, 1.2, 'a', "aa", [1, 2, 3]]
```

— **En compréhension** : méthode concise pour traiter tout ou partie des éléments d'un itérable et renvoyer un dictionnaire contenant les résultats.

```
1 dico1 = {n : n ** 2 for n in range(20) if n%2==0}
2 cles = [1, 2, 3]
3 valeurs = ["un", "deux", "trois"]
4 dico2 = {k: v for (k, v) in zip(cles, valeurs)}
```

La première instruction génère un dictionnaire contenant des clés n liées à leurs valeurs n^2 si n est pair. La seconde génère un dictionnaire à partir de deux listes. Elle utilise la fonction `zip` qui prend deux ou plusieurs séquences et retourne une liste de tuples. Chaque tuple contient un élément de chaque séquence.

1.2 Fonctions en Python

Il est fréquent d'utiliser des **fonctions** en Python : cela simplifie l'écriture du code, en permettant l'appel de la fonction au moment opportun. Le squelette de la fonction nommée `toto` est le suivant :

```
def toto(parametres):
    instruction 1
    instruction finale
    return valeur
```

Cette structure appelle les commentaires suivants :

- il ne faut pas oublier le caractère `:` à la fin de la première ligne définissant la fonction ;
- on peut mettre autant de paramètres que l'on veut dans la définition de la fonction ;
- on peut ne pas mettre l'instruction `return`, la fonction renvoie alors l'objet `none` ;
- les variables définies à l'intérieur de la fonction ne sont pas visibles ou utilisables à l'extérieur : ce sont des **variables locales** !
- pour afficher le `return` de la fonction on peut utiliser l'instruction `print`, comme dans les exemples ci-dessous.

Voici un exemple de fonction et son résultat :

```
# fonction moyenne arithmetique de 3 valeurs
def moyenne(x,y,z):
     m=(x+y+z)/3.0
     print('affichage du calcul : ',m)
     return m
print('finalement ',moyenne(10,12,15))
```

et le résultat :

```
('affichage du calcul : ', 12.333333333333334)
('finalement ', 12.333333333333334)
```

On propose de façon similaire une définition récursive du calcul de la factorielle d'un entier :

```
# fonction factorielle recursive
def fact(n):
    if n<=1:
         return 1
    else:
         return n*fact(n-1)
a=5
print('la factorielle de ',a,' est', fact(a))
```

et l'affichage :

```
('la factorielle de ', 5, ' est', 120)
```

1.3 Librairies usuelles

Il existe plus de 200 modules ou librairies en Python. Parmi les plus utiles :

— le module `math` qui permet d'avoir accès à toutes les constantes et fonctions mathématiques, autrement en nombre très réduit ;

— le module `random` qui permet de générer des nombres pseudo-aléatoires selon de multiples distributions ;

— le module `matplotlib` qui permet de faire des graphiques, par exemple tracer une liste de données en fonction d'une autre (contenant le même nombre de données) ;

— le module `numpy` qui permet d'effectuer tout type de calculs numériques ;

— le module `scipy`, basé sur le module `numpy`, qui permet d'avoir accès à des fonctions et classes de haut niveau, permettant de traiter des données, un peu comme avec les logiciels MATLAB ou Scilab.

Nous donnons ci-dessous un exemple permettant de faire un graphe à partir d'une fonction mathématique. La figure 1.1 montre le résultat.

```
1  # importation des librairies et des fonctions utiles
2  # ici affichage plot et les fonctions sin et cos
3  import matplotlib.pyplot as plt
4  import numpy as np
5  # remplissage de 3 listes (x,y,z)
6  x = np.arange(0.0, 6.0,0.1)
7  y = np.sin(x)
8  z=np.cos(2*x)
9  # affichage des 2 graphes
10 plt.plot(x,y,label='sin(x)')
11 plt.plot(x,y,'b+')
12 plt.plot(x,z,label='cos(2x)')
13 plt.plot(x,z,'ro')
14 # affichage des legendes et textes
15 plt.xlabel('abscisse')
16 plt.ylabel('ordonnee')
17 plt.title('une tentative de graphe')
18 plt.grid(True)
19 plt.legend()
20 # sauvegarde figure en PNG
21 plt.savefig("graphe.png")
22 plt.show()
```

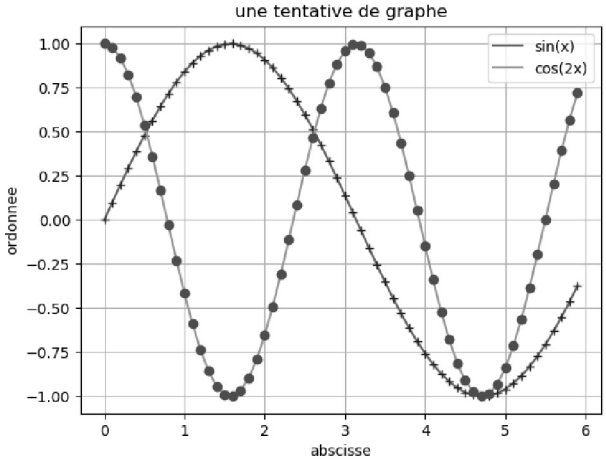


FIGURE 1.1 – Librairies graphiques et mathématiques

Le tableau qui suit présente quelques indications sur les paramètres graphiques de **matplotlib pyplot** :

couleur	bleue	rouge	noir	cyan	vert
	b	r	k	c	g
forme	point	étoile	plus	croix	rond
	.	*	+	x	o
trait	continu	discontinu	point		
	-	- -	:		

1.4 Librairies pour l'IoT

Nous le traiterons dans un chapitre suivant, mais un des protocoles de communication entre machines parmi les plus utilisés dans l'IoT est MQTT.

Voici un exemple de message MQTT transmis par un capteur installé dans une salle de cours qui émet de la donnée via le protocole LoRaWAN, ici au format JSON :

```
1  {
2    "applicationID":"1",
3    "applicationName":"AM107",
4    "deviceName":"AM107-11",
5    "deviceProfileName":"Milesight",
6    "deviceProfileID":"0b68c925-7e17-46ae-98e5-bf3093808a6d",
7    "devEUI":"24e124128c012114",
8    "rxInfo":[],
9    "txInfo":{"frequency":868300000,"dr":5},
10   "adr":true,
```

```
  "fCnt":21449,
  "fPort":85,
  "data":"A2fDAARoYgZlBAAIAAMABWoAAAd9swEIfewACXMDJg==",
  "object":
  {
    "activity":0,
    "co2":435,
    "humidity":49,
    "illumination":4,
    "infrared":3,
    "infrared_and_visible":8,
    "pressure":973.1,
    "temperature":19.5,
    "tvoc":236
  }
}
```

Comme on le voit ci-dessus, c'est un objet de type dictionnaire `python`. Le code suivant donne un exemple dans utilisant la libraire `paho-mqtt`. C'est un client MQTT Python qui va s'abonner à un flux et extraire des messages une grandeur particulière :

```
import paho.mqtt.client as mqtt
import json

# config
mqttServer = "chirpstack.iut-blagnac.fr"
appID = "1"
deviceID = "+" # Tous les devices

topic_subscribe = "application/{}/device/{}/event/up".format(appID,
    deviceID)

print("On commence...")

# callback appelle lors de la reception d'un message
def get_data(mqttc, obj, msg):
    jsonMsg = json.loads(msg.payload)
    print("Date/heure envoie paquet : ")
    print(jsonMsg["rxInfo"][0]['time'])
    donnee = jsonMsg["object"]

    try:
        print(donnee["co2"])
        co2 = donnee["co2"]
        print("La concentration de CO2 est :{}".format(co2))
        if co2 > 700:
            print("il va falloir aérer !")
        else:
```

```
27              print("ok, tout va bien ! ")
28      except json.JSONDecodeError:
29          print("Erreur décodage ?")
30
31  mqttc = mqtt.Client(transport="websockets")
32  mqttc.connect(mqttServer, port=1884, keepalive=60)
33
34  mqttc.on_message = get_data
35
36  # soucription au device
37  mqttc.subscribe(topic_subscribe, 0)
38
39  mqttc.loop_forever()
```

1.5 Exercices

Exercice 1 : Calcul du PGCD
On donne le code en Python ci-dessous qui est censé déterminer le PGCD des deux entiers c et d.

1. Exécuter le code à la main et vérifier qu'il contient une erreur.
2. Corriger cette erreur et calculer le PGCD de c et d.

```
1  # exercice 1.1
2  # Entiers c>0 et d>0 dont on cherche le PGCD
3  d = 4563
4  c = 195
5  if c < d :
6           c,d  = d,c
7  c0=c;
8  d0=d;
9  while d == 0:
10          c,d = d,c % d
11 print('Le PGCD de', c0, 'et', d0, 'vaut', c)
```

Solution

1. le résultat obtenu est que le PGCD de 4563 et 195 est censé être 4563, ce qui est bien entendu faux.
2. L'erreur commise dans le programme se situe à la ligne 9 : il faut remplacer `==` par `!=` sinon la boucle `while` n'est pas effectuée. La solution est donc :

```
1  # exercice 1.1 : solution
2  # Entiers c>0 et d>0 dont on cherche le PGCD
3  d = 4563
4  c = 195
```

```
5  if c < d :
6  c,d  = d,c
7  c0=c;
8  d0=d;
9  while d != 0:
10 c,d = d,c % d
11 print('Le PGCD de', c0, 'et', d0, 'vaut', c)
```

Dans ce cas, le PGCD de 4593 et 195 est 39.

Exercice 2 : La factorielle

On donne le code en Python ci-dessous qui doit en principe remplir la liste `factorielle` avec les valeurs successives de $n!$

1. Combien la liste contient-elle d'éléments ?
2. Quels sont ces éléments ?
3. Montrer qu'il y a une erreur grave dans ce code, la corriger et contrôler le résultat.

```
1  # exercice 1.2
2  #  initialisation
3  N=6
4  factorielle=[]
5  # boucles imbriquees
6  for j in range(1,N+1):
7           fact= 1
8           for i in range(1, j+1):
9               fact = fact * i
10              factorielle.append(fact)
11 # affichage de la longueur de la  liste  avec len()
12 print('la liste contient : ',len(factorielle), ' elements')
13 # affichage du contenu de la  liste
14 print('les elements de la liste sont : ',factorielle)
```

Solution

1. Il y a 21 éléments dans la liste nommée `factorielle`.
2. Ces éléments sont :

$$1, 1, 2, 1, 2, 6, 1, 2, 6, 24, 1, 2, 6, 24, 120, 1, 2, 6, 24, 120, 720$$

3. Si le code était correct, on aurait trouver seulement 6 valeurs dans la liste : $1, 2, 6, 24, 120, 720$. Le problème du code proposé vient de l'indentation de la ligne 10. Le code corrigé est :

```
1  # exercice 1.2
2  #  initialisation
3  N=6
```