# *Chapitre 2*
# Les Fonctions

Ce chapitre est le deuxième chapitre de l'année de Première NSI. Il introduit les **fonctions**, un concept central de la programmation : des blocs d'instructions nommés et réutilisables. On y apprendra à définir ses propres fonctions, à utiliser les fonctions prédéfinies de Python, à importer des bibliothèques et à réaliser des simulations.

## I. Introduction

Comment un informaticien fait-il bouillir de l'eau avec une casserole vide, un robinet et une plaque chauffante ? Il remplit la casserole, il fait chauffer la casserole jusqu'à ébullition. Et comment un informaticien fait-il bouillir de l'eau avec une casserole vide, un robinet et une plaque chauffante ? Il vide la casserole et ré-applique la réponse de la question 1 !

En informatique, on essaie d'être assez fainéant mais productif : lorsque l'on a fait quelque chose, on essaie de le réutiliser. Pour ce faire, on utilise des fonctions. Comme en mathématiques, *une fonction reçoit quelque chose en entrée et donne quelque chose en sortie*. En informatique, parfois, ce quelque chose c'est rien.

**Définition 2.1.**
Une **fonction** est un bloc d'instructions nommé, qui peut recevoir des valeurs en entrée (ses **paramètres**) et renvoyer une valeur en sortie. Une fois définie, elle peut être **appelée** et réutilisée autant de fois que nécessaire, avec des valeurs différentes.

## II. Les fonctions en Python

### 1. Généralités

En Python, il est possible de définir des fonctions. Les fonctions en programmation permettent d'utiliser de multiples fois une séquence d'instructions qui dépend d'un certain nombre de paramètres (on parle aussi d'arguments). La définition d'une fonction passe par :

— la définition du **nom** de la fonction ;
— la définition de sa **liste de paramètres** en entrée (qui peut être vide) ;
— la définition de la **séquence d'instructions** dans la fonction ;

— la définition de la **sortie** de la fonction (qui est facultative en Python).
Elle prend la forme suivante :

```
def nom_fonction(a, b, c):
    instructions
    return sortie
```

Ici, la première ligne comporte :
— la commande `def`, qui indique qu'une fonction va être définie ;
— `nom_fonction`, qui désigne le nom de la fonction (choisi par l'utilisateur) ;
— des variables `a`, `b` et `c`, qui sont les paramètres de la fonction (leur nom est choisi par l'utilisateur) ; ils sont placés entre parenthèses et séparés les uns des autres par des virgules (il peut y en avoir n'importe quel nombre entier, y compris 0, auquel cas on écrit `nom_fonction()`) ;
— un deux-points « : ».

Cette première ligne est suivie, après indentation, de la séquence d'instructions de la fonction, qui se termine généralement (mais pas nécessairement) par un renvoi de la fonction indiqué par la commande `return`. Le renvoi permet de définir une valeur renvoyée en sortie. Le code d'une fonction peut contenir plusieurs renvois correspondant à différents cas (en utilisant des instructions conditionnelles, par exemple). Toutefois, l'exécution de la fonction n'effectue jamais qu'un seul renvoi.

**■ Exemple 2.1.**
La fonction `f` prend un paramètre $x$ et renvoie la valeur obtenue par l'instruction `2*x + 3`. En mathématiques, elle correspond à une fonction affine de la forme $f : x \mapsto 2x + 3$. ■

```
def f(x):
    return 2*x + 3
```

**■ Exercice 2.1.**
En utilisant la fonction `f` définie à l'exemple précédent, déterminer les valeurs renvoyées par les appels `f(3)`, `f(0)` et `f(10)`. ■

**■ Exemple 2.2.**
Un appel de fonction s'écrit avec le nom de la fonction suivi de ses arguments entre parenthèses. La valeur renvoyée peut être stockée dans une variable ou affichée avec `print`. ■

```
def f(x):
    return 2*x + 3

print(f(2))
print(f(5))
print(f(f(1)))
```

Ce programme affiche `7`, `13` puis `13`. En effet, `f(1)` vaut 5, donc `f(f(1))` = `f(5)` = 13. Remarquez que la fonction `f` est appelée trois fois avec des valeurs différentes : c'est tout l'intérêt des fonctions.

■ **Exemple 2.3.**
La fonction `puissance` prend deux paramètres $a$ et $n$ et renvoie en sortie $a^n$. On remarque dans le code suivant que l'indentation de la boucle s'ajoute à l'indentation de la définition de la fonction. ■

```
def puissance(a, n):
    p = 1
    for k in range(n):
        p = p * a
    return p
```

■ **Exemple 2.4.**
La fonction `mini` prend deux paramètres $a$ et $b$ et renvoie le plus petit des deux. Ici, le renvoi dépend d'une instruction conditionnelle. ■

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

■ **Exemple 2.5.**
La fonction `bonjour` affiche `Bonjour !` à l'écran. Elle ne prend aucun paramètre et ne fait aucun renvoi : ses parenthèses sont vides. ■

```
def bonjour():
    print("Bonjour !")
```

> **R Une fonction qui doit produire un résultat ne doit JAMAIS se contenter de l'afficher avec `print` (0 sinon !)** Une fonction qui calcule doit **renvoyer** son résultat avec `return` : c'est cette valeur renvoyée qui pourra être réutilisée dans la suite du programme. Afficher un résultat et renvoyer un résultat sont deux choses différentes.

■ **Exemple 2.6.**
Comparons une fonction qui affiche son résultat et une fonction qui le renvoie : ■

```
def f_affiche(x):
    print(x * 2)
```

```
def f_renvoie(x):
    return x * 2

print(f_renvoie(4))
```

Ce programme affiche `8`. En revanche, si l'on écrit `print(f_affiche(4))`, on obtient d'abord l'affichage de `8` (provoqué par le `print` interne à la fonction), puis l'affichage de `None` : la fonction `f_affiche` ne renvoie rien, sa valeur est `None`.

**Exemple 2.7.**
Une fonction peut calculer une valeur dans une variable interne, puis renvoyer cette variable : ■

```
def volume_boite(L, l, h):
    v = L * l * h
    return v

print(volume_boite(2, 3, 4))
```

Ce programme affiche `24`. La variable `v` n'existe qu'à l'intérieur de la fonction : on dit qu'elle est **locale** à la fonction.

**Exemple 2.8.**
Le code d'une fonction peut contenir plusieurs `return`, correspondant à différents cas. Toutefois, une seule exécution de la fonction n'effectue qu'un seul renvoi : celui du premier `return` rencontré. ■

```
def signe(x):
    if x > 0:
        return "positif"
    elif x < 0:
        return "negatif"
    else:
        return "nul"
```

**Exemple 2.9.**
Une fonction peut appeler une autre fonction. C'est une façon de réutiliser du code déjà écrit : ■

```
def carre(x):
    return x ** 2

def somme_carres(a, b):
    return carre(a) + carre(b)

print(somme_carres(3, 4))
```

Ce programme affiche `25` : on calcule `carre(3)` $= 9$, `carre(4)` $= 16$, puis $9 + 16 = 25$.

> **R** Avant d'écrire une fonction, on peut décrire ce qu'elle fait : son **entête** (nom et paramètres), ce qu'elle **renvoie** et ce qu'elle **calcule**. Cette description s'appelle la **spécification** de la fonction. Elle permet de savoir exactement à quoi s'attendre, et de tester la fonction avec des exemples.

## 2. Fonctions prédéfinies

Dans Python, un certain nombre de fonctions sont déjà prédéfinies. Elles peuvent donc être utilisées directement sans avoir à les redéfinir. Quelques-unes de ces fonctions ont déjà été évoquées dans le cours précédent (`print`, `input`). Le tableau suivant contient une liste de fonctions prédéfinies en Python qui pourront vous servir dans le cadre de ce cours. Cette liste n'est bien évidemment pas exhaustive et vous pourrez découvrir les autres fonctions prédéfinies en Python en consultant la documentation disponible via le lien suivant : https://docs.python.org/3/.

| Fonction | Description |
|---|---|
| `print(x)` | Affiche la valeur de `x` dans la console |
| `input(txt)` | Affiche le texte de la chaîne de caractères `txt` dans la console ; l'utilisateur doit alors taper une valeur qui sera renvoyée par la fonction `input` dès qu'il appuie sur la touche Entrée du clavier |
| `int(x)` | Renvoie la valeur de `x` convertie en entier lorsque cela est possible |
| `float(x)` | Renvoie la valeur de `x` convertie en flottant lorsque cela est possible |
| `len(x)` | Renvoie le nombre d'objets (ou de caractères) de `x` |
| `abs(x)` | Renvoie la valeur absolue de `x` |

**▪ Exemple 2.10.**
Dans le code suivant, on utilise les fonctions `float`, `input`, `print` et `round` : ▪

```
x = float(input("Veuillez rentrer une valeur pour x"))
print(f"La valeur arrondie de x vaut {round(x)}")
```

Remarquez l'encapsulation de la fonction `input` dans la fonction `float`. Ceci est nécessaire car la fonction `input` ne permet que de récupérer le texte tapé par l'utilisateur (même si ce texte comporte des nombres). Ainsi, si l'on n'utilise pas la fonction `float` et que l'utilisateur rentre la valeur `123.3` avant de valider par la touche Entrée, c'est la chaîne de caractères `"123.3"` qui sera stockée dans la variable

`x`. La fonction `float` permet alors de convertir la chaîne de caractères `"123.3"` en `123.3` qui est un nombre flottant. La fonction `round`, utilisée dans cet exemple, renvoie l'entier le plus proche d'un nombre.

**Exercice 2.2.**
On exécute le programme suivant.

```
x = float(input("Veuillez rentrer une valeur pour x"))
```

1. Que contient la variable `x` si l'utilisateur tape `12.5` puis appuie sur la touche Entrée ?
2. Que se passe-t-il si l'on remplace `float` par `int` dans ce programme et que l'utilisateur tape `12.5` ?

■

**Exemple 2.11.**
Les fonctions `int`, `float` et `str` permettent de convertir une valeur d'un type vers un autre : ■

```
print(int("42") + 1)
print(float("3.14") + 1)
print(str(25) + " ans")
```

Ce programme affiche `43`, `4.14` puis `25 ans`. Sans conversion, l'expression `"42" + 1` aurait provoqué une `TypeError`.

**Exemple 2.12.**
La fonction `len` renvoie le nombre de caractères d'une chaîne ; la fonction `abs` renvoie la valeur absolue d'un nombre : ■

```
print(len("bonjour"))
print(abs(-7))
```

Ce programme affiche `7` puis `7` : `"bonjour"` contient 7 caractères et la valeur absolue de $-7$ vaut 7.

# III. Bibliothèques

## 1. Importer des bibliothèques

En plus des fonctions prédéfinies qui sont accessibles directement, Python propose également un certain nombre de bibliothèques (aussi appelées modules) qui contiennent de nombreuses fonctions déjà programmées (ainsi que d'autres objets prédéfinis). Pour utiliser les fonctions d'une bibliothèque, il faut d'abord importer la bibliothèque. La manière la plus simple pour ce faire est d'utiliser la commande `import` suivi du nom de la bibliothèque.

■ **Exemple 2.13.**
L'instruction suivante permet d'importer la bibliothèque standard `math`. ■

```
import math
```

Pour utiliser une fonction d'une bibliothèque (ou n'importe quel autre type d'objet d'une bibliothèque), il suffit alors d'utiliser la syntaxe `nom_bibliotheque.nom_fonction`.

■ **Exemple 2.14.**
Le code suivant permet d'importer la bibliothèque `math` dans Python pour utiliser la fonction `sqrt`, qui permet de calculer une racine carrée. Ici, on affiche la valeur de $\sqrt{2}$. ■

```
import math
print(math.sqrt(2))
```

Une autre façon d'importer les fonctions et objets d'une bibliothèque est d'utiliser une instruction suivant la syntaxe ci-dessous :

```
from nom_bibliotheque import *
```

Cette instruction permet d'importer toutes les fonctions et objets d'une bibliothèque de manière à ce qu'ils puissent être utilisés sans rappeler le nom de la bibliothèque.

■ **Exemple 2.15.**
Après l'importation `from math import *`, on peut écrire directement `sqrt(2)` sans le préfixe `math.` : ■

```
from math import *
print(sqrt(2))
```

## 2. Fonctions aléatoires

Le langage Python permet également la définition de **fonctions aléatoires**. En informatique, une fonction aléatoire est une fonction dont la valeur renvoyée est le résultat d'un tirage aléatoire. Cette valeur n'est donc pas déterminée de manière unique par la valeur des paramètres en entrée de la fonction. La bibliothèque `random` en Python contient un certain nombre de fonctions aléatoires prédéfinies et notamment la fonction `randint`, qui renvoie un entier compris entre deux valeurs fournies en paramètres, de manière aléatoire et selon une loi de probabilité uniforme.

■ **Exemple 2.16.**
Dans le code suivant, on importe les fonctions de la bibliothèque `random` afin de tirer au hasard soit 0, soit 1. ■

```
import random as rd
print(rd.randint(0, 1))
```

L'instruction `import random as rd` importe la bibliothèque `random` en lui donnant le diminutif `rd` : on écrit alors `rd.randint` au lieu de `random.randint`. C'est une simple question de commodité d'écriture.

■ **Exercice 2.3.**
Quelles valeurs la fonction `rd.randint(2, 6)` peut-elle renvoyer ? Combien y a-t-il de valeurs possibles ? ■

■ **Exemple 2.17.**
Simulons le lancer d'un dé équilibré à six faces : chaque appel renvoie un entier entre 1 et 6. ■

```
import random as rd
face = rd.randint(1, 6)
print(face)
```

### Simulation

Une **simulation** est un processus qui calque un phénomène réel dans le but d'en prédire l'issue. Le résultat d'une simulation peut ainsi informer sur le résultat possible du phénomène. On utilise généralement les simulations en informatique pour recréer un phénomène qui coûterait trop cher à réaliser physiquement (comme pour une simulation de crash-test d'avion) ou qui est tout simplement impossible à réaliser physiquement (comme pour une simulation de l'évolution climatique du globe terrestre).

■ **Exemple 2.18.**
On peut utiliser la fonction `randint` comme précédemment pour simuler le lancer d'une pièce qui peut tomber sur pile (que l'on fait correspondre au 0) ou face (que l'on fait correspondre au 1). Si un lancer de pièce est facile à réaliser physiquement, le lancer d'un million de pièces par exemple devient plus compliqué alors qu'en simulation cela ne met que quelques secondes sur un ordinateur moderne. Dans le code Python suivant, une simulation permet de comptabiliser combien de fois une pièce tombe sur face en un million de lancers. C'est la variable `p` qui permet de stocker cette valeur. ■

```
import random as rd
p = 0
for k in range(1000000):
    p = p + rd.randint(0, 1)
```

À la fin de l'exécution, la variable `p` contient le nombre de faces obtenues. Comme la probabilité d'obtenir face vaut $\frac{1}{2}$ à chaque lancer, ce nombre est en pratique proche de 500000.

# IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

### Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : calcul mental d'expressions Python, lecture de code court, prédiction de renvoi de fonction, complétion d'en-têtes. Chaque question doit pouvoir être traitée en moins de trente secondes.

**■ Exercice 2.4.**
Types, valeurs et premiers renvois : répondre aux questions suivantes sans exécuter les programmes.

1. Quel est le type de la valeur renvoyée par chacune des deux expressions suivantes ?

```
9 / 2
9 // 2
```

2. Que renvoie l'appel `triple(4)` avec la fonction définie ci-dessous ?

```
def triple(x):
    return x * 3
```

3. Que renvoie l'appel `produit(6, 7)` avec la fonction définie ci-dessous ?

```
def produit(a, b):
    return a * b
```

4. Donner la valeur de l'expression Python suivante.

```
"ab" * 3
```

5. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
7.0 + 3
```

■

**■ Exercice 2.5.**
Affectations, calculs et appels de fonctions : déterminer ce qui est affiché par chacun des programmes suivants.

1. Qu'affiche le programme suivant ?

```
n = 3
n = n + 5
```

```
n = n * 2
print(n)
```

2. Qu'affiche le programme suivant ?

```
def incremente(x):
    return x + 10

print(incremente(5))
```

3. Qu'affiche le programme suivant ?

```
a = 4
b = a * 2
a = 100
print(b)
```

4. Qu'affiche le programme suivant, où les appels sont imbriqués ?

```
def double(x):
    return x * 2

print(double(double(5)))
```

5. Qu'affiche le programme suivant ?

```
def carre(x):
    return x ** 2

print(carre(5))
```

■

■ **Exercice 2.6.**

Booléens, conditions et en-têtes de fonctions : répondre aux questions suivantes.

1. Indiquer si chacune des expressions suivantes vaut `True` ou `False`.

```
(2 < 5) and (3 > 7)
(6 > 2) or (9 < 3)
not (3 != 3)
```

2. Quelle valeur prend la variable `categorie` après exécution du programme suivant ?

```
age = 12
if age < 13:
    categorie = "enfant"
elif age < 18:
    categorie = "adolescent"
else:
```

```
    categorie = "adulte"
```

3. Écrire l'en-tête d'une fonction `aire` qui prend deux paramètres `longueur` et `largeur`.
4. Écrire l'en-tête d'une fonction `saluer` qui ne prend aucun paramètre.
5. Quelle est l'erreur dans l'en-tête suivant ?

```
def calcul(x y):
    return x + y
```

■

■ **Exercice 2.7.**

Boucles, `return` manquant et affichage : répondre aux questions suivantes.

1. Combien de fois le corps de la boucle suivante est-il exécuté ?

```
for i in range(0, 30, 5):
    pass
```

2. Qu'affiche le programme suivant, où la fonction ne renvoie rien ?

```
def carre(x):
    resultat = x * x

print(carre(5))
```

3. Qu'affiche le programme suivant, où la fonction affiche au lieu de renvoyer ?

```
def double(x):
    print(x * 2)

resultat = double(5)
print(resultat)
```

4. Quelle valeur est affichée par le programme suivant ?

```
compteur = 0
while compteur < 5:
    compteur = compteur + 2
print(compteur)
```

5. Recopier et compléter le corps de la fonction `perimetre` pour qu'elle renvoie le périmètre d'un carré de côté `cote`.

```
def perimetre(cote):
    return 4 * ____
```

■

■ **Exercice 2.8.**
Boucles non bornées et portée des variables : répondre aux questions suivantes.

1. Quelle valeur est affichée par le programme suivant ?

```
n = 1
while n < 50:
    n = n * 3
print(n)
```

2. Qu'affiche le programme suivant ?

```
x = 7

def afficher():
    x = 2
    print(x)

afficher()
print(x)
```

3. Le programme suivant provoque-t-il une boucle infinie ? Expliquer la réponse en une phrase.

```
n = 10
while n != 0:
    n = n - 3
```

4. Qu'affiche le programme suivant ?

```
def f():
    a = 4
    return a

def g():
    a = 5
    return a

print(f() + g())
```

5. Que renvoie l'appel `est_pair(9)` avec la fonction définie ci-dessous ?

```
def est_pair(n):
    if n % 2 == 0:
        return True
```

■

## Exercices d'application

Les exercices d'application reprennent les notions du chapitre, du plus simple au plus difficile. Ils demandent d'écrire de petites fonctions Python.

■ **Exercice 2.9.**
Écrire une fonction `aire_rectangle(longueur, largeur)` qui renvoie l'aire d'un rectangle dont on donne la longueur et la largeur. Indication : l'aire vaut longueur $\times$ largeur. Tester la fonction avec les valeurs 5 et 3 : le programme doit afficher `15`. ■

■ **Exercice 2.10.**
Écrire une fonction `maximum(a, b)` qui renvoie le plus grand des deux nombres `a` et `b`. On utilisera une instruction conditionnelle. Tester la fonction avec les appels `maximum(3, 8)` et `maximum(10, 2)` : on doit obtenir `8` puis `10`. ■

■ **Exercice 2.11.**
Écrire une fonction `est_pair(n)` qui renvoie `True` si l'entier `n` est pair et `False` sinon. Indication : `n` est pair si `n % 2 == 0`. Tester la fonction avec `est_pair(10)` et `est_pair(7)`. ■

■ **Exercice 2.12.**
Écrire une fonction `celsius_en_fahrenheit(c)` qui renvoie la température en degrés Fahrenheit correspondant à `c` degrés Celsius. On rappelle la formule : $F = C \times \frac{9}{5} + 32$. Tester la fonction avec `celsius_en_fahrenheit(0)` : on doit obtenir `32.0`. ■

■ **Exercice 2.13.**
Écrire une fonction `repeter(mot, n)` qui renvoie la chaîne de caractères formée en répétant `n` fois le mot `mot`. Indication : l'opérateur `*` permet de répéter une chaîne de caractères. Tester la fonction avec `repeter("ha", 3)` : on doit obtenir `"hahaha"`. ■

■ **Exercice 2.14.**
Écrire une fonction `somme(n)` qui renvoie la somme des entiers de 1 à `n` à l'aide d'une boucle bornée. Tester la fonction avec `somme(5)` : on doit obtenir `15`. ■

■ **Exercice 2.15.**
Le tableau suivant donne la mention en fonction de la note $N$ obtenue à la première session du baccalauréat.

| Condition | Mention |
|---|---|
| $0 \leq N < 8$ | Ajournée |
| $8 \leq N < 10$ | Rattrapage |
| $10 \leq N < 12$ | Sans Mention |
| $12 \leq N < 14$ | Assez Bien |
| $14 \leq N < 16$ | Bien |
| $N \geq 16$ | Très Bien |

Définir une fonction Python `mention(note)` qui prend une note en paramètre et renvoie la mention correspondante sous forme de chaîne de caractères. Attention aux bornes des intervalles ! Tester la fonction avec les notes 7, 9, 11, 13, 15 et 17.
■

### Exercices de réflexion

Les exercices de réflexion demandent plus d'analyse : déroulement de programmes, attention aux indentations, repérage d'erreurs, modélisation d'un problème. Ils sont classés du plus abordable au plus exigeant.

■ **Exercice 2.16.**
Déterminer les renvois des fonctions `f` et `g` définies ci-dessous pour les appels `f(2)`, `f(0)`, `g(1)` et `g(3)`.

```
def f(x):
    y = x**2 - 3
    return y

def g(x):
    y = x**2 - 3
    z = 3*x - y
    y = z + x
    return z
```

■

■ **Exercice 2.17.**
On considère les trois fonctions Python suivantes.

Détailler les calculs de `f1(5, 1)`, `f2(5, 1)` et `f3(5, 1)`. On listera l'ensemble de toutes les affectations qui sont réalisées en indiquant si celles-ci ont lieu :

— avant le début de la boucle ;
— dans la boucle et, dans ce cas, au combien de tours de la boucle ;
— après la sortie de la boucle.

```
def f1(x, y):
    z = 0
    while z < 20:
        z = x + y
        x = 2 * x
        y = 3 * y
        z = y * z
    return z

def f2(x, y):
    z = 0
    while z < 20:
        z = x + y
        x = 2 * x
        y = 3 * y
    x = y * z
    return z

def f3(x, y):
    z = 0
    while z < 20:
        z = x + y
    x = 2 * x
    y = 3 * y
    return z
```

■

■ **Exercice 2.18.**
On souhaite écrire une fonction qui échange les valeurs de deux variables. Léa propose le code suivant.

```
def echange(a, b):
    a, b = b, a

x = 3
y = 8
echange(x, y)
```

```
print(x, y)
```

1. Qu'affiche ce programme ? Justifier la réponse.
2. Expliquer pourquoi la fonction de Léa ne remplit pas son objectif.
3. Proposer une version corrigée du programme.

■

■ **Exercice 2.19.**

Le programme suivant est censé calculer la moyenne de deux nombres, puis afficher cette moyenne augmentée de 1. Il contient plusieurs erreurs.

```
def moyenne(a, b):
    m = a + b / 2
    print(m)

x = moyenne(4, 6)
print(x + 1)
```

1. Sans l'exécuter, expliquer ce que fait réellement ce programme et préciser le type de chaque erreur (erreur de logique, `TypeError`, ...).
2. Écrire une version corrigée du programme.

■

■ **Exercice 2.20.**

1. Recopier et compléter le code suivant de la fonction `lancer` pour qu'elle simule le lancer d'un dé équilibré à six faces.

```
import random as rd

def lancer():
    return rd.randint(..., ...)
```

2. On souhaite réaliser une simulation du lancer de 1000 dés. Écrire un programme, à la suite du code précédent, qui appelle la fonction `lancer` 1000 fois, compte le nombre d'apparitions de chaque face, puis affiche ces effectifs. On pourra utiliser six variables `nb1`, `nb2`, ..., `nb6` initialisées à 0.

■

■ **Exercice 2.21.**

Écrire une fonction `factorielle(n)` qui renvoie le produit $1 \times 2 \times 3 \times \cdots \times n$ (noté $n!$) à l'aide d'une boucle bornée. Indication : initialiser une variable `p` à 1 avant la boucle, puis multiplier cette variable à chaque tour. Tester la fonction avec `factorielle(5)` : on doit obtenir `120`. ■