

♦ TP1 – Rappels sur le langage Python (I)

I. — Variables, opérations et comparaisons

1) Variable et affectation

Une variable est un espace mémoire désigné par une étiquette (un nom). Lorsqu'on appelle une variable nommée **A** dans une instruction (par exemple dans l'addition **A+3** ou dans la comparaison **A > 0**), l'instruction s'effectue en fait sur la valeur affectée à la variable **A** c'est-à-dire à la valeur stockée dans l'espace mémoire étiqueté par **A**.

Affecter une valeur **v** à une variable **A**, c'est stocker la valeur **v** dans l'espace mémoire étiqueté par **A**. L'affectation d'une valeur **v** à une variable **A** se fait de la manière suivante :

en langage naturel	en Python
$A \leftarrow v$	A = v

Remarque. Lorsqu'on affecte la variable **A**, soit elle existait déjà et alors son ancienne valeur est supprimée et remplacée par la nouvelle valeur affectée soit la variable **A** est créée avec la valeur affectée.

ATTENTION ! En Python, l'ordre est essentiel lors de l'affectation. Pour affecter la valeur **v** à la variable **A**, on ne peut pas écrire **v = A**.

Il est possible de faire plusieurs affectations simultanées sous la forme **A, B, C = v, w, x**.

Il est également possible d'échanger les valeurs de deux variables **A** et **B** (ou plus) en utilisant la syntaxe **A, B = B, A**.

2) Opérations sur les nombres et types **int** et **float**

Python permet d'effectuer des opérations sur les nombres à l'aide des opérateurs suivants : + (addition), - (soustraction), * (multiplication), / (division) et ** (puissance).

En Python, les données sont typées c'est-à-dire que leurs valeurs sont associées à un certain type. Il existe deux types de nombres : les entiers (type **int**) et les nombres flottants, c'est-à-dire les nombres « à virgule » (type **float**). Rappelons qu'en Python le séparateur numérique n'est pas la virgule mais le point. Ainsi, **A = 3** affecte l'entier 3 à la variable **A** alors que **A = 3.0** ou **A = 3.** affecte le nombre flottant 3 à la variable **A**. Dans ces deux cas, la valeur de **A** est la même (3) mais le type est différent.

L'addition, la soustraction, la multiplication et la puissance (entièrre) d'entiers renvoient un entier. Tout autre opération renvoie un flottant. Ainsi,

3+4 renvoie l'entier 7,	3*2.5 renvoie le flottant 7.5,
4.6+1.25 renvoie le flottant 5.85,	12/3 renvoie le flottant 4.0,
3+2.6 renvoie le flottant 5.6,	3/4 renvoie le flottant 0.75
3.5+1.5 renvoie le flottant 5.0,	4.5/1.5 renvoie le flottant 3.0,
3*4 renvoie l'entier 12	2**3 renvoie l'entier 8,
4.3*2.5 renvoie le flottant 10.75	2.5**2 renvoie le flottant 6.25.

Il est possible de convertir des entiers en flottants et inversement à l'aide des fonctions **float** et **int**. Par exemple, **float(145)** renvoie le flottant 145.0. Attention, en revanche, **int(35.67)** renvoie l'entier 35 et **int(-217.345)** renvoit l'entier -217.

Le typage a une importance cruciale lorsqu'on fait des opérations sur les variables car certaines opérations ne sont possibles que sur des variables du même type ou car une même opération donnera des résultats différents selon le type des variables. Il existe des opérateurs spécifiques aux entiers : `//` qui renvoie le quotient dans la division euclidienne (c'est-à-dire la division exacte avec reste) et `%` qui renvoie le reste dans la division euclidienne. Ainsi, `17//3` et `17%3` renvoient respectivement 5 et 2 car $17 = 5 \times 3 + 2$ donc, dans cette division, le quotient est 5 et le reste est 2.

Remarque. Pour connaître le type d'une donnée ou d'une variable, on peut utiliser la fonction `type` qui renvoie `<class 'int'>` pour entier et `<class 'float'>` pour un flottant.

3) Comparaisons et type `bool`

En Python, on peut effectuer des comparaisons à l'aide des opérateurs suivants.

Tester si	$a = b$	$a \neq b$	$a < b$	$a > b$	$a \leq b$	$a \geq b$
syntaxe Python	<code>a == b</code>	<code>a != b</code>	<code>a < b</code>	<code>a > b</code>	<code>a <= b</code>	<code>a >= b</code>

Ce type de test renvoie `True` (vrai) ou `False` (faux) suivant que la comparaison est vraie ou fausse. Ce résultat fait partie d'un nouveau type, appelé type booléen (`bool`).

On peut effectuer des comparaisons plus complexes en enchaînant plusieurs comparaisons grâce aux opérateurs `or` (ou), `and` (et) et `not` (contraire de).

Ainsi, si on effectue les affectations `A = 3` et `B = -2` alors `A != 0` renvoie `True`, `A == B` renvoie `False`, `A >= 3` renvoie `True`, `A > 0 or B > 0` renvoie `True`, `A > 0 and B > 0` renvoie `False` et `not A == B+5` renvoie `False`.

4) Exercices

Exercice 1. Pour chacune des instructions suivantes, déterminer le résultat renvoyé ainsi que le type de celui-ci.

- | | | |
|---|--|---------------------------------------|
| 1) $4+3$ | 2) $4+3.$ | 3) $4*3.$ |
| 4) $3/4$ | 5) $20//3$ | 6) $20\%3$ |
| 7) $3 > 1$ | 8) $3 == 2+1$ | 9) $3**3$ |
| 10) $3.**2$ | 11) <code>not 4 > 5</code> | 12) $2 > 1 \text{ and } 3 > 4$ |
| 13) $2 > 1 \text{ or } 3 > 4$ | 14) $17\%2 == 1$ | 15) <code>not (2**3 > 3**2)</code> |
| 16) $(3 != 1+1) \text{ and } (\text{not } 3 > 4)$ | 17) <code>not (2 == 1 \text{ or } 2 > 1)</code> | 18) $1.2+3.8 != 5$ |

Exercice 2. Pour chacun des algorithmes suivants, déterminer la valeur des différentes variables à la fin de l'exécution. Traduire ensuite l'algorithme en un programme Python.

1)

```

 $a \leftarrow 2$ 
 $b \leftarrow 3$ 
 $a \leftarrow a + 1$ 
 $b \leftarrow a + b$ 

```

2)

```

 $a \leftarrow 2$ 
 $b \leftarrow 3$ 
 $a \leftarrow a \times b$ 
 $b \leftarrow a^2$ 

```

3)

```

 $a \leftarrow 0$ 
 $b \leftarrow 1$ 
 $a \leftarrow b$ 
 $b \leftarrow a$ 

```

4)

```

 $a \leftarrow 1$ 
 $b \leftarrow 2$ 
 $a \leftarrow a + b$ 
 $a \leftarrow a \times b$ 
 $a \leftarrow a^b$ 

```

$$(a,b) = (b,a)$$

Exercice 3. Pour chacun des programmes suivants, déterminer la valeur des différentes variables à la fin de l'exécution.

1)

```
a = 2
b = 3
a = a + b
b = a + b
```

2)

```
a = 2
b = 3
a, b = b, a
b = 2*a+b**3
```

3)

```
a = 2
b = 3
c = 4
a, b, c = 3*c, 2*a, b
```

Exercice 4. Sans utiliser l'échange de variable, modifier le programme écrit dans la question 3) de l'exercice 2 de telle façon qu'à la fin de l'exécution, les valeurs de **a** et **b** soient échangées.

II. — Fonctions et instructions conditionnelles

1) Fonctions

Une fonction est une sorte sous-programme d'un programme Python dédié à réaliser une tâche particulière et auquel on peut faire appel dans le programme principal ou dans une autre fonction.

L'intérêt d'une fonction est d'éviter de réécrire plusieurs fois le même code d'instructions à différents endroits d'un programme.

La syntaxe est la suivante.

```
def nom_de_la_fonction(liste de paramètres):
    instructions
```

Si on veut que la fonction renvoie un résultat, on utilise la fonction **return**. L'exécution d'une fonction s'arrête dès le premier **return** rencontré.

Remarque. Pour que la syntaxe soit correcte en Python, il est indispensable que :

1. la ligne commençant par **def** (l'identificateur de fonction) se termine par deux points (:
2. le bloc d'instructions soit écrit en retrait par rapport aux autres lignes. Ce retrait est appelé l'indentation.

Voici un exemple. La fonction **somme** suivante possède deux paramètres **a** et **b**.

```
def somme(a, b):
    c = a + b
    return(c)
```

L'instruction **somme(3,5)** renvoie la somme de 3 et 5 c'est-à-dire l'entier 8. Dans ce cas, on dit qu'on a appelé la fonction **somme** en passant 3 et 5 en arguments. Ainsi, des arguments sont des valeurs données aux paramètres de la fonction (ici, **a** et **b**).

Il existe un certain nombre de fonctions pré-existantes en Python. Citons, en particulier, la fonction **print** qui permet d'afficher à l'écran les quantités passées en argument et **input** qui permet de demander à l'utilisateur de saisir une ou plusieurs valeurs.

Ainsi, si on veut voir afficher à l'écran la valeur d'une variable **A**, on écrit **print(A)** et, de même, si on veut afficher à l'écran la valeur renvoyée par la fonction **somme** lorsqu'on passe 3 et 5 en arguments, on écrit **print(somme(3,5))**.

2) Instruction conditionnelle

Une instruction conditionnelle est une instruction du type

Si (condition C) alors (bloc d'instructions 1) sinon (bloc d'instructions 2)

ce qui signifie : si la condition C est vraie, effectuer le bloc d'instructions 1 et sinon (si elle est fausse), effectuer le bloc d'instructions 2.

Remarque. La partie « sinon (bloc d'instructions 2) » n'est pas obligatoire. Si on l'omet alors aucune action n'est effectuée si la condition C est fausse.

La syntaxe d'une instruction conditionnelle est la suivante.

en langage naturel	en Python
Si (condition C) bloc d'instructions 1 Sinon bloc d'instructions 2	<pre>if (condition C): instructions 1 else: instructions 2</pre>

Voici un exemple. La fonction `lpg` prend en arguments deux nombres (entiers ou flottants) et renvoie le plus grand des deux.

```
def lpg(a,b):  
    if a >= b:  
        return(a)  
    else:  
        return(b)
```

Remarque. Comme pour les fonctions, les deux points à la fin de la ligne contenant `if` et après `else` ainsi que l'indentation pour les blocs d'instructions 1 et 2 sont indispensables. Par ailleurs, Lorsqu'on a plus de deux cas possibles dans une instruction conditionnelle, on utilise la syntaxe suivante :

```
if (condition 1):  
    instructions 1  
elif (condition 2):  
    instructions 2  
else:  
    instructions 3
```

On peut ajouter autant de conditions intermédiaires que l'on veut en ajoutant `elif` devant chacune d'elles.

3) Exercices

Exercice 5. Écrire une fonction `difference` telle que `difference(a,b)` renvoie $a-b$ et une fonction `produit` telle que `produit(a,b)` renvoie $a \cdot b$.

Exercice 6.

1. Écrire une fonction `est_isocèle` qui prend en arguments trois nombres entiers `a`, `b` et `c` et qui renvoie `True` si le triangle de côtés `a`, `b` et `c` est isocèle et `False` sinon.
2. Écrire de même une fonction `est_rectangle` qui prend en arguments trois nombres entiers `a`, `b` et `c` et qui renvoie `True` si le triangle de côtés `a`, `b` et `c` est rectangle et `False` sinon.

Exercice 7. Écrire une fonction `lpp` prenant en arguments deux nombres `a` et `b` et qui renvoie le plus petit des deux.

Exercice 8. Écrire une fonction `valeur_absolue` prenant en argument un nombre `x` et qui renvoie la valeur absolue de `x`.

Exercice 9. Écrire une fonction `est_entier` qui prend en argument un nombre (entier ou flottant) `x` et qui renvoie `True` si la valeur de `x` est entière (indépendamment du type) et `False` sinon. Ainsi, `est_entier(3)` ou `est_entier(3.0)` renvoie `True` et `est_entier(4.23)` renvoie `False`.

Indication. En Python, `3 == 3.0` renvoie `True`.

Exercice 10. Écrire une fonction `est_pair` prenant en argument un entier `n` et qui renvoie `True` si `n` est pair et `False` sinon.

Indication. On pourra utiliser l'opérateur `%`.

Exercice 11.

1. Écrire une fonction `intervalle1` prenant en argument un nombre `x` et qui renvoie `True` si `x` appartient à l'intervalle $]-2; 3]$ et `False` sinon.
2. Écrire une fonction `intervalle2` prenant en argument un nombre `x` et qui renvoie `True` si `x` appartient à l'ensemble $]-\infty; -3] \cup [5; +\infty[$ et `False` sinon.
3. Écrire une fonction `intervalle3` prenant en argument un nombre `x` et qui renvoie `True` si `x` appartient à l'ensemble $]-5; -3] \cup [0; 2[$ et `False` sinon.
4. Écrire une fonction `intervalle4` prenant en argument un nombre `x` et qui renvoie `True` si `x` est strictement positif et différent de 1 ou si `x` est strictement négatif et différent de -1 et `False` sinon.

Exercice 12. Écrire une fonction `signe` prenant un nombre `x` en argument et qui affiche `positif` si le nombre est strictement positif, `nul` si le nombre est nul et `négatif` si le nombre est strictement négatif.

Indication. Pour afficher `positif`, la syntaxe est `print("positif")`.

Exercice 13. Une année est bissextile si elle est divisible par 4 mais pas par 100 ou bien si elle est divisible par 400. Par exemple, 2024 est bissextile car 2024 est divisible par 4 mais pas par 100. De même, 2000 est bissextile car 2000 est divisible par 400. En revanche, 2100 n'est pas bissextile car 2100 est divisible par 100 mais pas par 400.

Écrire une fonction Python `est_bissextile` qui prend en argument un entier naturel `n` et affiche `bissextile` si l'année `n` est bissextile et non `bissextile` sinon.

III. — Boucles

1) Boucle bornée (boucle « Pour »)

Une boucle bornée (ou boucle « Pour ») est une boucle dans laquelle des instructions identiques sont répétées un nombre de fois déterminé à l'avance.

Pour répéter 10 fois un bloc d'instructions, la syntaxe est la suivante.

en langage naturel	en Python
Pour k variant de 0 à 9 bloc d'instructions Fin Pour	for k in range(10): instructions

Dans cette syntaxe, k est une variable dont le nom peut être choisi arbitrairement.

Remarque. Comme pour l'instruction conditionnelle, les deux points à la fin de la première ligne et l'indentation sont indispensables.

Il existe différentes syntaxes pour **for k in range(...)**:

- **for k in range(n):** : k est une variable de type **int** qui prend successivement les valeurs 0, 1, ..., $n - 1$.
- **for k in range(a,b):** : k est une variable de type **int** qui prend successivement les valeurs $a, a + 1, \dots, b - 1$.
- **for k in range(a,b,p):** : k est une variable de type **int** qui prend successivement les valeurs $a, a + p, a + 2p, a + 3p, \dots$, jusqu'à atteindre la plus grande valeur possible strictement inférieure à b (ici, le pas est donc p et non pas 1 comme dans la syntaxe précédente).

Par exemple, la fonction suivante affiche tous les entiers pairs de 0 à $2n$ pour un entier naturel n passé en argument.

```

def nb_pairs(n):
  for k in range(n+1):
    print(2*k)

```

Les boucles bornées sont particulièrement adaptées au calcul des termes d'une suite récurrente. Considérons, par exemple, la suite (u_n) définie par $u_0 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = 2u_n + r$. La fonction suivante renvoie la valeur de u_n pour un entier naturel n passé en argument.

```

def suite(n):
  u = 1
  for k in range(n):
    u = 2*u+k
  return u

```

2) Boucle non bornée (boucle « Tant que »)

Une boucle non bornée (ou boucle « Tant que ») est une boucle dans laquelle un bloc d'instructions est répété tant qu'une certaine condition C est vraie. La boucle s'arrête dès que la condition C est fausse.

La syntaxe est la suivante.

en langage naturel	en Python
Tant que (condition C) bloc d'instructions Fin Tant que	while condition C: instructions

Remarque. Comme pour une boucle bornée, les deux points à la fin de la première ligne et l'indentation sont indispensables. La condition C peut être n'importe quelle expression de type `bool`. Ce peut être une comparaison du type `A == 3` ou `A > 0` mais ce peut aussi être le résultat renvoyé par une fonction si celui-ci est un booléen.

Les boucles bornées sont particulièrement adaptées à la recherche de seuil, notamment pour les suites. Considérons, par exemple, la suite (u_n) définie par $u_0 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = \frac{u_n}{n+1}$. La fonction suivante renvoie la plus petite valeur de l'entier naturel n tel que $u_n \leq \text{eps}$ où eps est un flottant passé en argument.

```
def seuil(eps):
    u = 1
    n = 0
    while (u > eps):
        u = u/(n+1)
        n = n+1
    return n
```

La condition dans la boucle `while` est le contraire de ce qu'on cherche puisque le calcul continue tant qu'on n'a pas atteint de seuil voulu. On pourrait d'ailleurs remplacer la condition `(u > eps)` par `(not u <= eps)`. Dans cette fonction, n est une variable qui augmente de 1 à chaque tour de boucle, on dit qu'elle est incrémentée de 1 à chaque tour. On l'a ici implémenté par `n = n+1`. Une autre syntaxe possible est `n += 1`.

3) Exercices

Exercice 14. On considère l'algorithme suivant.

```
S ← 0
Pour i allant de 1 à 10
    S ← S + i2
Fin Pour
```

1. Quelle est la fonction de cet algorithme ?
2. Traduire cet algorithme en un programme Python puis implémenter ce programme.

Exercice 15. La fonction suivante est censée calculer la somme des puissances de 3 de 3^4 jusqu'à 3^n où n est un entier supérieur ou égal à 4 passé en argument.

```
def somme_puissance_trois(n):
    S = 0
    for j in range(4, n):
        S = S + 3^j
    return S
```

1. Il y a plusieurs erreurs dans ce programme. Les trouver et les corriger.
2. Implémenter cette fonction corrigée.

Exercice 16. Écrire une programme qui prend deux entiers naturels non nuls n et p en argument et qui renvoie la somme des puissances p -ième des entiers de 1 à n , c'est-à-dire $\sum_{k=1}^n k^p$.

Exercice 17.

- Écrire une fonction `mult_7` qui renvoie le nombre de multiples de 7 compris entre 1 et n , où n est un entier naturel non nul passé en argument.

Indication. Un entier n est un multiple de 7 si son reste dans la division euclidienne par 7 est nul.

- Écrire une fonction `mult_7_pas_3_5` qui prend en argument un entier naturel n non nul et qui renvoie le nombre d'entiers compris entre 1 et n qui sont divisibles par 7 mais pas par 3 ni par 5.

Exercice 18. On dit qu'un entier naturel n est parfait si la somme de ses diviseurs positifs est égale $2n$. Par exemple, 6 est parfait car les diviseurs positifs de 6 sont 1, 2, 3 et 6 et $1 + 2 + 3 + 6 = 12 = 2 \times 6$.

Écrire une fonction `est_parfait` qui renvoie le nombre d'entiers parfaits compris entre 1 et n , où n est un entier naturel non nul passé en argument.

Exercice 19. On souhaite écrire une fonction `factorielle` prenant en argument un entier naturel n et renvoyant la valeur de $n!$.

- Expliquer pourquoi la fonction suivante ne convient pas.

```
def factorielle(n):
    fact = 1
    for i in range(n+1):
        fact = fact*i
    return fact
```

- Corriger la fonction précédente pour obtenir le résultat voulu.

Exercice 20.

- Soit (u_n) la suite définie par $u_0 = 4$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = 2 - \frac{u_n}{2}$. Écrire une fonction `suite_u` qui renvoie le terme d'indice n de la suite (u_n) pour un entier naturel n passé en argument.
- Soit (F_n) la suite définie par $F_0 = 0$, $F_1 = 1$ et, pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$. Écrire une fonction `suite_F` qui renvoie le terme d'indice n de la suite (F_n) pour un entier naturel n passé en argument.

Exercice 21. On considère l'algorithme suivant.

```
A ← 1
Tant que A < 10
    Afficher A
    A = A + 2
Fin Tant que
```

- Quelle est la fonction de cet algorithme ?
- Traduire cet algorithme en un programme Python puis implémenter ce programme.

Exercice 22. On considère l'algorithme suivant.

```
k ← 0
Tant que  $k^2 < 1000$ 
    k ← k + 1
Fin Tant que
```

1. Quelle est la fonction de cet algorithme ?
2. Traduire cet algorithme en un programme Python puis implémenter ce programme.

Exercice 23. La fonction suivante est censée calculer la somme des nombres impairs inférieurs à n pour un entier naturel n passé en argument.

```
def somme_impairs(n):
    k = 1
    S = 0
    while (S < n)
        k += 2
        S = S+k
    return S
```

1. Il y a plusieurs erreurs dans la syntaxe de cette fonction. Les trouver et les corriger.
2. Implémenter cette fonction.

Exercice 24. Sur un compte en banque, on place 1000 €. Le compte rapporte 2% d'intérêt par an. Ainsi, au bout d'un an, il y aura sur le compte $1000 + \frac{2}{100} \times 1000 = 1020$ €.

Écrire une fonction `nb_ANNÉES` qui renvoie le nombre d'années nécessaires pour que le montant sur le compte dépasse strictement P € où P est un nombre strictement positif passé en argument. On suppose que le taux reste à 2% et qu'on n'ajoute rien d'autre sur le compte que les intérêts annuels.

Exercice 25. Écrire une fonction `plus_gd_carré` qui renvoie le plus grand entier inférieur ou égal à n qui est le carré d'un entier, où n est un entier naturel passé en argument.

Exercice 26. Écrire une fonction `comptage` qui renvoie le nombre d'entiers naturels non nuls tels que k^k est inférieur ou égal à n où n est un entier naturel non nul passé en argument.

Exercice 27. Pour $n \in \mathbb{N}$, on pose $S_n = \sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} |i - j|$. Écrire une fonction `somme_diff_abs` qui

renvoie la valeur de S_n pour un entier naturel n non nul passé en argument. On pourra utiliser la fonction `valeur_absolue` de l'exercice 9.

Exercice 28. On considère la fonction `mystère` suivante.

```
def mystere(n):
    s = 0
    while n > 0:
        s = s+(n%10)
        n = n//10
    return(s)
```

1. Déterminer, sans l'implémenter, la valeur renvoyée par `mystère(1234)`.
2. Si n est un entier naturel, que représente la valeur renvoyée par `mystère(n)` ?
3. En s'inspirant de la fonction `mystère`, écrire une fonction `nb_chiffres` qui renvoie le nombre de chiffres d'un entier naturel n passé en argument.

IV. — Les modules

Il existe en Python de nombreuses fonctions prédéfinies qui ne sont pas automatiquement chargées mais qui sont stockées dans des fichiers appelés modules. Pour utiliser de telles fonctions, il faut au préalable importer le module qui les comporte.

Pour charger un module, on utilise la syntaxe suivante :

```
from nom_du_module import *
```

Il existe de très nombreux modules. Nous allons en évoquer seulement deux.

1) Le module `math`

Le module `math` contient toutes les constantes et les fonctions (au sens mathématiques) dont nous aurons besoin. On le charge à l'aide de la syntaxe suivante :

```
from math import *
```

Il contient en particulier

- `pi` : renvoie une valeur approchée de π à 10^{-16} près ;
- `sqrt(x)` : renvoie la racine carrée du nombre x ;
- `abs(x)` : renvoie la valeur absolue du nombre x ;
- `cos(x)` : renvoie le cosinus du nombre x (en radians) ;
- `sin(x)` : renvoie le sinus du nombre x (en radians) ;
- `e` : renvoie une valeur approchée du nombre e à 10^{-16} près ;
- `exp(x)` : renvoie l'image de x par la fonction exponentielle (i.e. le nombre e^x) ;
- `log(x)` : renvoie l'image de x par la fonction logarithme népérien (i.e. le nombre $\ln(x)$) ;
- `log10(x)` : renvoie l'image de x par la fonction logarithme décimal (i.e. le nombre $\log(x)$) .

2) Le module `random`

Le module `random` est un module qui permet d'engendrer des nombres pseudo-aléatoires. On le charge à l'aide de la syntaxe suivante :

```
from random import *
```

Il contient en particulier les fonctions :

- `random()` qui renvoie un nombre flottant pseudo-aléatoire entre 0 et 1 ;
- `randint(a,b)` qui renvoie un entier pseudo-aléatoire compris (au sens large) entre les deux entiers a et b .

3) Exercices

Exercice 29.

1. En utilisant la fonction `sqrt`, écrire une fonction `carre_parfait` qui renvoie `True` si l'entier n passé en argument est le carré d'un entier et `False` sinon.
2. En utilisant la fonction précédente, écrire une fonction `somme_carrés` qui renvoie la somme des carrés d'entiers compris entre 1 et n où n est un entier passé en argument. Ainsi, `somme_carré(10)` renvoie $1 + 4 + 9$ c'est-à-dire 14.

Exercice 30. Écrire une fonction `pile_ou_Face` qui renvoie au hasard et de manière équiprobable `pile` ou `face`.

Exercice 31. Écrire une fonction `lancer_de_dé` qui renvoie au hasard et de manière équiprobable un nombre entier entre 1 et 6 (compris).

Exercice 32. Lorsqu'on joue au Monopoly, on lance deux dés au hasard et on calcule la somme des valeurs obtenues.

En utilisant la fonction `lance_de_dé` de l'exercice 31, écrire une fonction `tirage_Monopoly` qui renvoie le résultat d'un tel tirage.

Exercice 33. En utilisant la fonction `lancer_de_dé` de l'exercice 31, écrire une fonction `simulation` prenant en argument un entier naturel n non nul, qui simule n lancers successifs d'un dé cubique équilibré et qui renvoie la fréquence de 6 obtenus (c'est-à-dire le nombre de 6 obtenus divisé par le nombre total de lancers).

Exercice 34. En utilisant la fonction `lancer_de_dé` de l'exercice 31, écrire une fonction qui simule une répétition de lancers de dé et renvoie le nombre de lancers nécessaires pour obtenir un 6 pour la première fois.

Exercice 35. La suite de Syracuse est une suite de nombres entiers définie par une valeur initiale u_0 et par la relation de récurrence :

$$\forall n \in \mathbb{N}, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Par exemple, en partant du nombre 7, on obtient la suite :

$$7, \quad 7 \times 3 + 1 = 22, \quad \frac{22}{2} = 11, \quad 11 \times 3 + 1 = 34, \quad \frac{34}{2} = 17, \dots$$

1. Écrire une fonction `syracuse` qui prend en arguments deux entiers naturels n et $a > 0$ et qui renvoie la valeur de u_n lorsque $u_0 = a$. On pourra utiliser la fonction `est_pair` de l'exercice 10.

2. Une conjecture (non encore démontrée à ce jour) postule que cette suite finit toujours par atteindre le nombre 1. La suite des valeurs de u_n partant de u_0 et allant jusqu'à la première apparition de 1 s'appelle un vol.

En utilisant la fonction `syracuse`, écrire une fonction `vol` qui prend en argument un entier $a > 0$ et qui renvoie toutes valeurs prises par (u_n) lors du vol lorsque $u_0 = a$.

3. On appelle temps de vol la plus petite valeur de n telle que $u_n = 1$. Écrire une fonction `temps_vol` prenant en argument un entier $a > 0$ et qui renvoie le temps de vol pour la suite (u_n) telle que $u_0 = a$.

4. Lors d'un vol, on appelle altitude maximale la plus grande valeur prise par la suite (u_n) . Écrire une fonction `altitude_max` qui prend en argument un entier $a > 0$ et qui renvoie l'altitude maximale pour la suite (u_n) telle que $u_0 = a$.