♦ TP1 – Rappels sur le langage Python (I)

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) `not 4 > 5`
12) `2 > 1 and 3 > 4`
13) `2 > 1 or 3 > 4`
14) `17%2 == 1`
15) `not (2**3 > 3**2)`
16) `(3 != 1+1) and (not 3 > 4)`
17) `not (2 == 1 or 2 > 1)`
18) `1.2+3.8 != 5`

Solution.
1) `4+3` renvoie l'entier `7`
2) `4+3.` renvoie le flottant `7.0`
3) `4*3.` renvoie le flottant `12.0`
4) `3/4` renvoie le flottant `0.75`
5) `20//3` renvoie l'entier `6` (car $20 = 3 \times 6 + 2$)
6) `20%3` renvoie l'entier `2` (voir ci-dessus)
7) `3 > 1` renvoie le booléen `True`
8) `3 == 2+1` renvoie le booléen `True`
9) `3**3` renvoie l'entier `27`
10) `3.**2` renvoie le flottant `9.0`
11) `not 4 > 5` renvoie le booléen `True`
12) `2 > 1 and 3 > 4` renvoie le booléen `False` (car $3 > 4$ est faux)
13) `2 > 1 or 3 > 4` renvoie le booléen `True` (car $2 > 1$ est vrai)
14) `17%2 == 1` renvoie le booléen `True` (car $17 = 2 \times 8 + 1$ donc le reste est bien égal à 1)
15) `not (2**3 > 3**2)` renvoie le booléen `True` (car $2^8 = 8 < 9 = 3^2$)
16) `(3 != 1+1) and (not 3 > 4)` renvoie le booléen `True` (car $3 \neq 2$ et $3 > 4$ est faux donc son contraire est vrai)
17) `not (2 == 1 or 2 > 1)` renvoie le booléen `False` (car $2 > 1$ est vrai donc la parenthèse est vraie et donc le contraire est faux)
18) `1.2+3.8 != 5` renvoie le booléen `False` (car le test d"égalité ne tient pas compte du type mais seulement de la valeur et $5{,}0 = 5$.)

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)

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

Solution. Pour chacun des algorithmes, on peut dresser une table d'exécution.

1)

a	b
2	
2	3
3	3
3	6

À la fin de l'exécution, a vaut 3 et b vaut 6. Le programme Python correspondant est :

```
a = 2
b = 3
a = a + 1
b = a + b
```

2)

a	b
2	
2	3
6	3
6	36

À la fin de l'exécution, a vaut 6 et b vaut 36. Le programme Python correspondant est :

```
a = 2
b = 3
a = a * b
b = a**2
```

3)

a	b
0	
0	1
1	1
1	1

À la fin de l'exécution, a et b valent 1. Le programme Python correspondant est :

```
a = 0
b = 1
a = b
b = a
```

4)

a	b
1	
1	2
3	2
6	2
36	2

À la fin de l'exécution, a vaut 36 et b vaut 2. Le programme Python correspondant est :

```
a = 1
b = 2
a = a + b
a = a * b
a = a**b
```

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
```

Solution. Dressons, là aussi, une table d'exécution.

1)

a	b
2	
2	3
5	3
5	8

À la fin de l'exécution, `a` contient l'entier `5` et `b` l'entier `8`.

2)

a	b
2	
2	3
3	2
3	14

À la fin de l'exécution, `a` contient l'entier `3` et `b` l'entier `14`.

3)

a	b	c
2		
2	3	
2	3	4
12	4	3

À la fin de l'exécution, `a` contient l'entier `12` et `b` l'entier `4` et `c` contient l'entier `3`.

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.

Solution. Une possibilité est d'introduire une nouvelle variables `c` pour conserver en mémoire la valeur de a :

```
a = 0
b = 1
c = a
a = b
b = c
```

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*b`.

Solution.

```
def difference(a,b):
  return a-b
```

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

Exercice 6.

1. Écrire une fonction `est_isocele` 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.

Solution.

1.

```
def est_isocele(a,b,c):
  if (a==b or a==c or b==c):
    return True
  else:
    return False
```

Autre possibilité (préférable)

```
def est_isocele(a,b,c):
  return a==b or a==c or b==c
```

2.

```
def est_rectangle(a,b,c):
  if (a**2+b**2==c**2 or a**2+c**2==b**2 or b**2+
     c**2==a**2):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
def est_rectangle(a,b,c):
  return a**2+b**2==c**2 or a**2+c**2==b**2 or b
     **2+c**2==a**2
```

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

Solution.

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

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

Solution.

```
def valeur_abslue(x):
  if (x>=0):
    return x
  else:
    return -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`.

Solution.

```
def est_entier(x):
  if (x == int(x)):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
def est_entier(x):
  return x == int(x)
```

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 `%`.

Solution.

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

Autre possibilité (préférable) :

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

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 -1 et `False` sinon.

Solution.

1.

```
def intervalle1(x):
  if (x>-2 and x<=3):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
def intervalle1(x):
  return x>-2 and x<=3
```

2.

```
def intervalle2(x):
  if (x<=-3 or x>=5):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
def intervalle2(x):
  return x<=-3 or x>=5
```

3.

```
def intervalle3(x):
  if ((x>-5 and x<=-3) or (x>=0 and x<2)):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
def intervalle3(x):
  reuturn (x>-5 and x<=-3) or (x>=0 and x<2)
```

Remarque. — Les parenthèses dans la condition

```
(x>-5 and x<=-3) or (x>=0 and x<2)
```

ne sont pas indispensables car en Python il y a priorité du `and` sur le `or` donc

```
x>-5 and x<=-3 or x>=0 and x<2
```

est automatique interprété comme `((x>-5 and x<=-3) or (x>=0 and x<2))`. Cependant, les parenthèses rendent le code plus lisible.

4.

```
def intervalle4(x):
  if ((x>0 and x!=1) or (x<0 and x!=-1)):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
def intervalle4(x):
  return (x>0 and x!=1) or (x<0 and x!=-1)
```

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")`.

Solution.

```
def signe(x):
  if (x>0):
    print('positif')
  elif (x==0):
    print('nul')
  else:
    print('négatif')
```

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.

Solution.

```
def est_bissextile(n):
  if ((n%4==0 and n%100!=0) or (n%400==0)):
    print('bissextile')
  else:
    print('non bissextile')
```

Exercice 14. On considère l'algorithme suivant.

S $\leftarrow$ 0
Pour i allant de 1 à 10
  S $\leftarrow$ S + i^2
Fin Pour

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

Solution.

1. L'algorithme proposé calcule la somme des carrés des entiers compris entre 1 et 10.
2. Le programme Python correspondant est

```
S=0
for i in range(1,11):
  S = S+i**2
```

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.

Solution.

1. Il y a une erreur de syntaxe : il faut remplacer `^` par `**` dans la quatrième ligne pour calculer la puissance. Il y a, de plus, une erreur dans le `range` : la variable `j` doit varier de 4 à n donc il faut écrire `range(4,n+1)`.
2. Le programme corrigé est donc

```
def somme_puissance_trois(n):
  S=0
  for j in range(4,n+1):
    S = S+3**j
  return S
```

Exercice 16. Écrire une fonction 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 $\sum_{k=1}^{n} k^p$.

Solution.

```
def somme_puissances(n,p):
  S=0
  for k in range(1,n+1):
    S = S + k**p
  return S
```

Exercice 17.

1. 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.
2. 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.

Solution.

1.

```
def mult_7(n):
  compteur = 0
  for k in range(1,n+1):
    if (k%7 == 0):
      compteur += 1
  return compteur
```

2.

```
def mult_7_pas_3_5(n):
  compteur = 0
  for k in range(1,n+1):
    if (k%7 == 0 and k%3 != 0 and k%5 != 0):
      compteur += 1
  return compteur
```

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.

Solution.

```
def est_parfait(n):
  compteur = 0
  for k in range(1,n+1)
    S=0
    for d in range(1,k+1):
      if (k%d == 0):
        S=S+d
    if S == 2*k:
      compteur += 1
  return compteur
```

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

1. Expliquer pourquoi la fonction suivante ne convient pas.

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

2. Corriger la fonction précédente pour obtenir le résultat voulu.

Solution.

1. La fonction proposée ne convient pas car `i` varie à partir de `0` donc, au premier tour de boucle Pour, `fact` prend la valeur `0` et, ensuite, elle reste constamment égale à `0`.
2.

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

Exercice 20.

1. 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.
2. 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.

Solution.

1.

```
def suite_u(n):
  u = 4
  for k in range(n):
    u = 2 - u/2
  return u
```

2.

```
def suite_F(n):
  F = 0
  G = 1
  for k in range(n):
    H = G
    G = G + F
    F = H
  return F
```

Autre possibilité :

```
def suite_F(n):
  F = 0
  G = 1
  for k in range(n):
    F, G = G, G + F
  return F
```

Exercice 21. On considère l'algorithme suivant.

A $\leftarrow$ 1
Tant que A $<$ 10
  Afficher A
  A = A + 2
Fin Tant que

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

Solution.

1. Cet algorithme affiche les entiers naturels impairs strictement inférieurs à 10.
2. Le programme Python correspondant est :

```
A = 1
while (A < 10):
  print(A)
  A += 2
```

Exercice 22. On considère l'algorithme suivant.

$k \leftarrow 0$
Tant que $k^2 < 1000$
  $k \leftarrow k + 1$
Fin Tant que

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

Solution.

1. Cet algorithme détermine le plus petit entier naturel dont le carré dépasse 1000.
2. Le programme Python correspondant est

```
k = 0
while (k**2 < 1000):
  k += 1
```

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.

Solution.

1. Il y a un erreur de syntaxe : il manque les deux points à la fin de la ligne `while` et deux erreurs de programmation : la condition est `k < n` et non pas `S < n`. De plus, il faut échanger les deux lignes après le `while` car on modifie la valeur de `k` avant de l'ajouter à `S` alors qu'il faut faire l'inverse.
2. Une correction possible est donc

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

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.

Solution.

```
def nb_années(P):
  C = 0
  M = 1000
  while (M <= P):
    M = M+(2/100)*M
    C += 1
  return C
```

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.

Solution.

```
def plus_gd_carré(n):
  k = 0
  while (k**2 <= n):
    k += 1
  return (k-1)**2
```

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

Solution.

```
def comptage(n):
  k = 1
  while (k**k <= n):
    k += 1
  return k
```

Exercice 27. Pour $n \in \mathbb{N}$, on pose $S_n = \sum_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant n}} |i - j|$. 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.

Solution.

```
def somme_diff_abs(n):
  S = 0
  for i in range(1, n+1):
    for j in range(1, n+1):
      S = S + valeur_absolue(i-j)
  return S
```

Exercice 28. On considère la fonction `mystere` 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 `mystere(n)` ?
3. En s'inspirant de la fonction `mystere`, écrire une fonction `nb_chiffres` qui renvoie le nombre de chiffres d'un entier naturel n passé en argument.

Solution.

1. Utilisons un table d'exécution :

s	0	4	7	9	10
n	1234	123	12	1	0
n > 0	True	True	True	True	False

   La valeur renvoyée par `mystere(1234)` est 10.

2. Si $n \in \mathbb{N}$, `mystere(n)` renvoie la somme des chiffres de n.

3.
```
def nb_chiffres(n):
  if n == 0:
    return 1
  else:
    nb = 0
    while n > 0:
      n = n//10
      nb += 1
    return nb
```

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.

Solution.

1.

```
from math import *

def carre_parfait(n):
  if (int(sqrt(n)) == sqrt(n)):
    return True
  else:
    return False
```

Autre possibilité (préférable) :

```
from math import *

def carre_parfait(n):
  return int(sqrt(n)) == sqrt(n)
```

2.

```
def somme_carrés(n) :
  S = 0
  for k in range(n+1) :
    if carre_parfait(k) :
      S = S + k
  return S
```

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

Solution. Pour simuler un tirage de *pile* ou *face*, on effectue un tirage aléatoire de 0 ou 1 et on associe 0 à pile et 1 à face. (Ce choix est arbitraire, on pourrait très bien faire le contraire, évidemment).

```
from random import *

def pile_ou_face():
  a = randint(0,1)
  if (a == 0):
    return "pile"
  else:
    return "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).

Solution.

```
from random import *

def lancer_de_dé():
  return randint(1,6)
```

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.

Solution.

```
def tirage_Monopoly():
  return lancer_de_dé()+lancer_de_dé()
```

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).

Solution.

```
def simulation(n):
  c=0
  for k in range(n):
    if (lancer_de_dé() == 6):
      c += 1
  return c/n
```

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.

Solution.

```
def premier_six():
  c=1
  while (lancer_de_dé() != 6):
    c+=1;
  return c
```

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 :

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

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

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

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 $u_0 = a$.

Solution.

1.

```
def syracuse(n,a):
  u = a
  for k in range(n):
    if est_pair(u):
      u = u//2
    else:
      u = 3*u+1
  return u
```

2.

```
def vol(a) :
  u = a
  n = 0
  while u  != 1 :
    print(u)
    n += 1
    u = syracuse(n,a)
  print(u)
```

3.

```
def temps_vol(a):
  u = a
  t = 0
  while u != 1:
    if est_pair(u):
      u = u//2
    else:
      u = 3*u+1
    t += 1
  return t
```

4.

```
def altitude_max(a):
  u = a
  m = a
  while u != 1:
    if est_pair(u):
      u = u//2
    else:
      u = 3*u+1
    if u > m:
      m = u
  return m
```