TP INFO 1 | Variables, tests, boucles et fonctions

Python a été créé en 1990 par Guido van Rossum, grand fan des Monty Python. C'est un langage orienté objet, impératif et interprété (le code écrit par le programmeur est transformé « à la volée » en langage machine exécutable par l'ordinateur). Sa syntaxe, éloignée de celle des langages de plus bas niveau, permet une initiation aisée aux concepts de base de la programmation. Python est très largement utilisé : Youtube, Google (Rossum a travaillé pour Google jusqu'au milieu des années 2000), la Nasa, Industrial light & Magic, etc. Il est de plus en plus utilisé dans les cours d'informatique.

1 Variables, tests, boucles et fonctions 1
1 Les environnements de développement 2
2 La notion de type en programmation 2
2.1 Entiers de Python 3
2.2 Flottants de Python 3
2.3 Booléens de Python 4
3 Les variables informatiques 5
4 Les tests 6
5 Les deux types de boucle 7
6 Les fonctions en informatique 8
7 Gestion des variables utilisées dans une fonction 9
8 Les modules de Python 10
9 Exercices 11
10 Indications 13

1. Les environnements de développement

Un environnement de développement est un logiciel permettant de rationaliser la création et le développement de programmes.

Il comporte un éditeur de texte permettant d'écrire et de sauvegarder des programmes longs et modulaires, des raccourcis pour les compiler ou les déboguer et un interpréteur.

Les environnements de développemnt suivants sont gratuits et compatibles avec Python : *Eclipse*, *Idle*, *Spyder*, *pyzo*, etc. Le lycée Louis-Le-Grand a choisi d'installer l'environnement de développement *pyzo* dans les salles de TP.

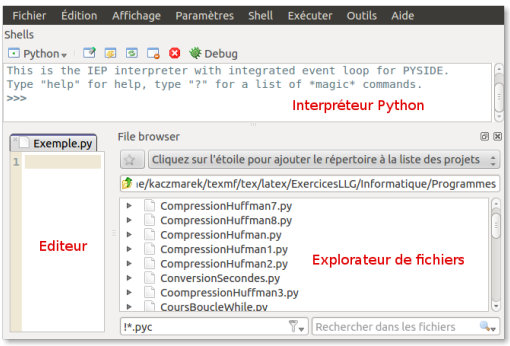
Pyzo est composé de trois fenêtres (qui peuvent se fermer et être redimensionnées) :

⇛ en haut, l'interpréteur Python ;

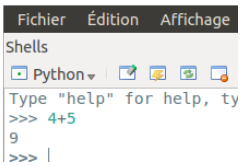
⇛ en bas à gauche, l'éditeur de texte ;

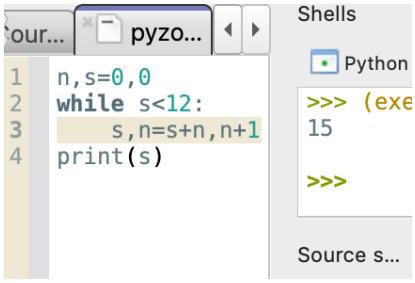
⇛ en bas à droite, l'explorateur de fichiers.

Le code peut être écrit dans l'interpréteur ou dans l'éditeur, son mode d'exécution en dépendera.



L'interpréteur s'utilise comme une calculatrice et, en cela, n'est adapté qu'à des vérifications, des programmes ou des calculs courts qui n'ont pas besoin d'être sauvegardés : on écrit une instruction qui est immédiatement exécutée en appuyant sur la touche *return*.





Les programmes plus longs ou destinés à être sauvegardés sont écrits dans l'éditeur et exécutés dans l'interpréteur (cf. le menu `Exécuter` dans la barre outil tout en haut de la fenêtre principale : on peut exécuter – *run* in english – tout le fichier ou seulement une sélection effectuée à la souris).

On utilise dans ce cas la fonction `print` afin de forcer l'affichage d'un résultat dans l'interpréteur.

Dans l'éditeur, tout ce qui suit le caractère # sera ignoré lors de l'exécution.

Ceci permet l'écriture de commentaires sur le code afin de l'expliquer et d'indiquer la signification des variables.

```
# Exercice 1
n,s=0,0                 # ceci est
while s<12:             # aussi un
    s,n=s+n,n+1         # commentaire
```

2. La notion de type en programmation

Dans un ordinateur, l'information est manipulée sous forme binaire quelque soit *sa nature* (nombre entier relatif, décimal, chaîne de caractères, etc). Par exemple, selon l'interprétation qu'on en fait, la séquence binaire `00111010` peut désigner le caractère de ponctuation : (selon le code ASCII, *American Standard Code for Information Interchange*) ou l'entier 58 écrit en base 2. Pour lever toute ambiguité sur l'interprétation d'une donnée sous forme binaire, il est nécessaire de lui associer un *type*, ie une interprétation. Nous exposerons dans un premier temps les types primitifs de Python.

2.1. Entiers de Python

Python utilise le type `int` pour les *entiers relatifs*. Ils sont manipulés en base 10, avec un signe – pour les éléments de $\mathbb{Z} \setminus \mathbb{N}$. Le nombre de chiffres est théoriquement illimité [1].

Opérateur	Opération
`+`	Addition
`-`	Soustraction
`*`	Multiplication
`/`	Quotient usuel
`a%b`	Reste dans la division euclidienne de a par b Se lit « a modulo b »
`a//b`	Quotient dans la division euclidienne de a par b
`a**n`	a puissance n

Attention à ne pas confondre `a//b` (quotient dans division euclidienne) et `a/b` (quotient dans $\mathbb{Q}$).

Il suffit d'écrire l'opération souhaitée dans une ligne active de l'interpéteur Python et d'appuyer sur la touche ENTRÉE pour obtenir le résultat :

```
>>> 45-4
41
>>> 45+34
79
```

```
>>> 45*4
180
>>> 45**2
2025
```

```
>>> 45//4
11
>>> 45%4
1
```

Plus généralement, en combinant des constantes et des opérateurs, on obtient ce qu'on appelle *une expression entière*.

La syntaxe est celle des mathématiques (utilisation de parenthèses, on ne divise par zéro, etc), nous ne l'expliciterons pas. Par exemple, `34`, `47-65*3` et `32-3165` sont des expressions entières.

2.2. Flottants de Python

Les *nombres à virgule flottante*, appelés aussi *flottants*, forment un sous-ensemble de l'ensemble des nombres décimaux. Ils permettent la représentation informatique des nombres réels. Nous les étudierons plus en détail dans le cours sur la représentation informatique des nombres. Contrairement au cas des entiers, la précision est ici limitée. Pour écrire un flottant, on utilise le symbole `.` au lieu de la virgule. On peut aussi utiliser la notation `m e n` pour $m10^n$ (m est la mantisse et n est l'exposant).

```
>>> 0.000457895433
0.000457895433
```

```
>>> 4.57895433e-4
0.000457895433
```

Les opérateurs `+`, `-`, `/`, `*` et `**` sont valables pour les nombres flottants. Voici quelques exemples depuis un interpéteur Python :

```
>>> 7.6*4.7
35.72
```

```
>>> 6.6-4.5
2.09999999999
```

```
>>> 6.5+9.5
16.0
```

On définit comme au paragraphe précédent la notion d'expression flottante (cf. page 3).

1. Python manipule ce qu'on appelle des entiers longs depuis la version 3. En fait, il y a évidemment une limite, la taille de la mémoire de l'ordinateur.

La précision étant limitée, les calculs sur les flottants, contrairement aux calculs sur les entiers, ne sont approchés et les propriétés usuelles de + et × (commutativité et associativité par exemple) ne sont plus vérifiées en général (cf. ci-dessus). La présence d'un flottant dans une expression numérique forcera le résultat à être du type flottant.

```
>>> 1+2345678990
2345678991
```

```
>>> 1.0+2345678990
2345678991.0
```

2.3. Booléens de Python

Le type `bool` (pour *booléens*) n'admet que deux constantes : `False` et `True`. Ce type correspond aux valeurs de vérité `Vrai` et `Faux` de la logique.

Pour combiner des booléens booléennes, on dispose des trois *opérateurs logiques*, *la disjonction*, *la conjonction* et *la négation* : `or` (ou), `and` (et), `not` (non). Ils sont définis par les tables de vérité suivantes :

b1	b2	b1 or b2	b1 and b2
True	True	True	True
True	False	True	False
False	True	True	False
False	False	False	False

b	not b
True	False
False	True

On étend naturellement la notion d'expression aux booléens (cf. page 3) au moyen de ces opérateurs. Voici une illustration dans un interpréteur Python :

```
>>> not(False or True)
False
```

```
>>> (False or True) and (not(True))
False
```

Les opérateurs de comparaison permettent de construire des valeurs booléennes à partir d'autres expressions. On peut appliquer à des couples d'expressions les opérateurs de comparaison suivants :

Opérateur	Signification
==	= (égalité)
!=	≠
>, <	>, <
>=, <=	⩾, ⩽

Voici quelques exemples de comparaison obtenues depuis un interpréteur Python :

```
>>> 1<2
True
```

```
>>> 2>=2
True
```

```
>>> 2==5-3
True
```

Ces opérateurs sont *paresseux*. Par exemple, dans `x and y`, si la valeur de l'expression `x` après évaluation est `False`, la deuxième expression `y` n'est pas évaluée.

3. Les variables informatiques

Une variable est une association entre un espace physique de la mémoire, un nom et une valeur. La définition d'une variable s'effectue en Python au moyen de l'opérateur =, comme l'illustre ce qui suit :

```
>>> x=2
```

```
>>> 2*x
4
```

```
>>> id(x)
148684964
```

Dans ce qui précède, la fonction `id` de Python permet d'accéder à la référence[2] `148684964` qui permet au système de trouver l'adresse-mémoire correspondant à la variable `x`. On retiendra la syntaxe d'une affectation et les noms de variables admissibles :

Symbole d'affectation	=
Noms de variable	Formés de lettres, de chiffres, du caractère tiret bas _ et commencent toujours par une lettre

On étend la notion d'expression au cas des variables (cf. page 3). Pour évaluer une expression, il suffit de l'écrire dans un interpréteur et d'appuyer sur la touche `Entrée`.

```
>>> x=0
>>> x=17
```

```
>>> 12*x+17
221
```

STOP Affection Versus égalité

Une des erreurs les plus courantes chez le pythonneux débutant est de confondre le symbole d'affectation = et le comparateur logique ==.

```
>>> x=0
```

```
>>> x
0
```

```
>>> x==1
False
```

```
>>> x
0
```

Python propose une syntaxe spécifique pour mettre à jour une variable `x`.

Code	Action
`x=exp`	L'expression `exp` est évaluée *puis* le résultat est stocké dans la variable `x`
`x+=exp`	A le même effet que `x=x+exp` : l'expression `exp` est évaluée, puis sa valeur est ajoutée à celle de `x` et le tout enregistré dans `x`
`x-=exp`	A le même effet que `x=x-exp`

2. Les mécanismes d'allocation de mémoire ne sont pas un attendu du programme et varient beaucoup selon les langages. Sous Python, les variables contiennent en fait *les références, pas les valeurs*. La référence d'une variable est générée lors de l'interprétation du programme et associée à l'adresse de la mémoire où est *effectivement stockée la valeur* de la variable.

Quelques exemples dans un interpréteur Python :

```
>>> x=3
>>> y=x
>>> x=1000
>>> z=y**2+1
```

```
>>> x,y,z
(1000,3,10)
>>> x*y
3000
```

```
>>> x=3
>>> x+=2
>>> x
5
```

Moralité : les modifications de la variable `x` sont sans impact[3] sur la variable `y`.

Continuons avec *les affectations multiples*. Elles sont spécifiques à Python se généralisent à un nombre fini quelconque de variables.

Code	Action
`x1,x2=exp1,exp2`	Les expressions `exp1` et `exp2` sont évaluées *puis* les résultats sont respectivement affectés aux variables `x1` et `x2`.

```
>>> x=2
>>> y=3
```

```
>>> x,y=2*x+3*y,4*x
```

```
>>> x,y
(13,8)
```

4. Les tests

C'est l'instruction `if...then` qui permet de réaliser un test. En voici la syntaxe.

Soit `b` une expression booléenne , `p1` et `p2` deux instructions. Le sens de l'instruction ci-contre est le suivant : si après évaluation `b` vaut `True`, alors `p1` est exécutée, sinon Python exécute `p2`.

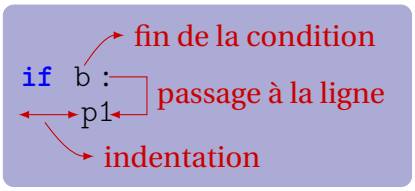
```
if b:
    p1
else:
    p2
```

```
if b:
    p1
```

L'alternative `else` dans le programme précédent est optionnelle. Le sens de l'instruction ci-contre est le suivant : si, après évaluation, `b` vaut `True`, alors l'instruction `p1` est exécutée ; sinon, il ne se passe rien.

STOP Respecter l'indentation

On respectera cette syntaxe en partie *spatiale* : *les deux points* « : » marquent la fin de la *condition du test* et *l'indentation* de `p1` est obligatoire, car délimite la portée de l'instruction `if`.



3. Attention, ce ne sera plus le cas pour des variables du type `list`. Cela vient essentiellement du fait que, en Python, l'affectation se fait par adressage et que les listes, contrairement aux entiers et aux flottants, est un type mutable.

Par exemple, le programme ci-contre affiche le plus grand des deux nombres a et b.

On peut généraliser cette syntaxe. On utilise alors le mot-clé `elif` (qui est la contraction de *else if*). L'instruction `else` permet de déterminer les autres cas.

```
if a<b:
    print(b)
else:
    print(a)
```

```
if b1:
    p1
elif b2:
    p2
etc.
elif bN:
    pN
else:
    p'
```

Dans ce cas, le déroulement de la suite d'instructions est la suivante :

- ⇛ L'expression booléenne `b1` est évaluée. Si elle est vraie, l'instruction `p1` est exécutée.
- ⇛ Si `b1` est fausse et `b2` est vraie, l'instruction `p2` est exécutée.
- ⇛ etc.
- ⇛ Si les `bk` pour $k \in [\![1, N]\!]$ sont fausses, l'instruction `p'` est exécutée.
- ⇛ À noter que dès qu'une condition est satisfaite, les instructions associées sont effectuées et le programme sort du test. Dans un test, on peut utiliser plusieurs instructions `elif`. Cependant, il y a au plus une instruction `else`.

5. Les deux types de boucle

La boucle while (*conditionnelle*) : Répéter un bloc d'instructions sous condition.

Voir ci-contre la syntaxe d'une boucle `while` (notez bien les deux points et l'indentation). Si l'expression booléenne `b` est `True`, on exécute l'instruction `p` et on revient au début de la boucle, on évalue à nouveau `b` ; sinon, on sort de la boucle.

```
while b:
    p
```

Une telle boucle est infinie lorsque l'expression booléenne `b` est toujours vraie. Pour interrompre une exécution, on cliquera sur l'icône d'arrêt de pyzo, voire on fermera puis relancera le shell.

La boucle for (*inconditionnelle*) : Répéter un bloc d'instructions un nombre de fois connu.

Ce code exécute `n` fois le bloc d'instructions `p`. Comme pour les tests et la boucle `while`, l'indentation est obligatoire. On peut choisir n'importe quel nom valable de variable à la place de `i`.

```
for i in range(n):
    p
```

```
s,i=0,1
while i<=4:
    s=s+i
    i=i+1
print(i,s)
```

```
s=0
for i in range(5):
    s=s+i
print(i,s)
```

La boucle `for` crée une variable `i` qu'elle *affecte* successivement aux valeurs 0, 1, 2, 3 et 4.

En fin de boucle, on obtient $(i, s) = (5, 10)$ dans le cas de la boucle `while`) et $(i, s) = (4, 10)$ pour la boucle `for`.

L'instruction `range(a,b)` génère $[\![a,b[\![$

On retiendra que `range` génère un intervalle fermé à gauche et ouvert à droite (principe général sous Python). Avec la syntaxe `range(n)`, l'indice initial vaut 0, on génère donc $[\![0,n-1]\!]$. Lorsque $a \geqslant b$, l'intervalle est vide. Notez qu'une boucle sur `range(3,1)` n'occasionne aucune erreur.

Cette syntaxe admet une généralisation : `range(a,b,r)` avec $r > 0$ et dans ce cas `i` décrit dans l'ordre croissant l'ensemble des entiers de la forme[4] $a + k \times r$ appartenant à $[\![a,b[\![$ où $k \in \mathbb{N}$. Par exemple, pour `range(3,17,2)`, la variable d'indice prend successivement les valeurs 3, 5, 7, 9, 11, 13, et 15.

```
for i in range(a,b,r):
    p
```

L'instruction `p` est executée pour chacune des valeurs `i` de la séquence générée par `range`.

6. Les fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous-programme, qui effectue une tâche auxiliaire clairement identifiée.

```
def mafonction(n):
    s=0
    for i in range(n+1):
        s=s+i**4
    return s
```

La fonction ci-contre calcule et renvoie le somme

$$0^4 + \cdots + n^4$$

pour tout entier n. La variable `n` est appelée *argument de la fonction*. Une fois définie (grâce à la commande `run`), tout utilisateur peut y faire appel en choisissant des valeurs numériques de n.

```
>>> mafonction(1)
1
```

```
>>> mafonction(2)
17
```

```
>>> mafonction(3)
98
```

Une fonction est considérée par Python comme un objet de type *function*.

En résumé, les deux principales manipulations sur les fonctions sont :

- ⇛ LA DÉCLARATION : étape qui permet de nommer la fonction et d'identifier le code qu'elle doit exécuter.
- ⇛ L'APPEL : étape où l'on exécute le code de la fonction pour des valeurs fixées de ses arguments.

La syntaxe pour définir (on dit aussi déclarer) une fonction est donnée ci-contre. Ne pas oublier : et l'*indentation* pour délimiter la fonction au sein du code.

```
def nom(arg1,...,argN):
    programme
```

On peut classer les fonctions en deux catégories[5] :

4. On reconnaît une progression arithmétique de premier terme a et de raison r. Il est possible de choisir $r < 0$ et dans ce cas, `i` décrit dans l'ordre décroissant l'ensemble des entiers de la forme $a + k \times r$ appartenant à $]\!]b,a]\!]$ où $k \in \mathbb{N}$.
5. Certains auteurs réservent les noms de *fonctions* aux premières et de *procédures* aux secondes.

⇛ celles qui *renvoient* une valeur (qui peut ensuite être utilisée pour de nouveaux calculs par exemple) ;

⇛ celles qui *ne renvoient rien*[6] (mais qui par exemple produisent de l'affichage, modifient certaines variables, etc).

C'est le mot-clé `return` qui permet de renvoyer une valeur dans une fonction. Dès que l'interpréteur *lit* `return(exp)`, l'exécution de la fonction se termine et la fonction renvoie l'expression `exp` ; la partie du code écrite après l'instruction `return` n'est jamais exécutée.

Pour bien comprendre l'utilisation de `return` au sein d'une fonction, nous donnons ci-dessous deux fonctions ; l'une *affiche* le maximum de deux nombres, l'autre *retourne* ce même maximum.

```
def maxprint(a,b):
    if a>b:
        print(a)
    else:
        print(b)
```

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

Dans l'interpréteur, exécuter `maxprint(2,3)` ou `max(2,3)` aboutit au même résultat, l'affichage de 3. Cependant, quand on y regarde de plus près, il y a une différence fondamentale : `max(2,3)` est une expression (qui peut donc être rétilisée dans d'autres calculs ou pour d'autres traitement), ce que `maxprint(2,3)` n'est visiblement pas.

```
>>> maxprint(2,3)
3
>>> maxprint(2,3)+2
Traceback (most recent call last): File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> max(2,3), max(2,3)+2
3,5
```

Signalons pour finir que, dans le langage Python, on peut passer une fonction en argument d'une autre fonction.

7. Gestion des variables utilisées dans une fonction

Une fonction ne peut modifier un de ses arguments si celui-ci est de type numérique. Nous verrons que ce n'est pas le cas des variables de type list ou ndarray (cf. le TP 2). La fonction ci-contre ne sert à rien : elle ne renvoie rien et ne peut modifier sa variable d'entrée.

```
def f(x):
    x=0
```

```
>>> a=1
>>> f(a)
>>> a
1
```

6. Sous Python, l'absence de `return` n'est qu'apparente car, dans ce cas, l'intepréteur ajoute en fait `return None` à la fonction.

Lors de l'appel d'une fonction, l'interpréteur Python crée des variables temporaires qui sont supprimées à la fin de l'appel. Ainsi, la portée des variables définies à l'intérieur de la fonction est limitée à la suite d'instructions définie à l'intérieur de celle-ci.

Les variables définies dans une fonction sont dites *locales*, par opposition aux variables *globales*.

```
def puiss2(n):
    N=2
    return n**N
```

```
>>> puiss2(10)
100
>>> N
```

```
Traceback (most recent call last):
File "<pyshell#27>", line 1, in <module> N
NameError: name 'N' is not defined
```

8. Les modules de Python

Un module est une collection de fonctions prédéfinies. Les fonctions usuelles des Mathématiques, telles que *cos* ou *abs* (la valeur absolue), sont prédéfinies. Afin d'utiliser une fonction prédéfinie, il faudra charger en mémoire un *module* la contenant. Pour le module `mod` et une fonction `f` de ce module, on utilisera la syntaxe suivante :

Code	Action
`import module`	Charge en mémoire le module `module`. On peut alors utiliser la fonction `f` sous la forme `module.f`
`import mod as m`	Charge en mémoire le module `mod` sous le pseudonyme `m`. On peut alors utiliser la fonction `f` sous la forme `m.f`
`from mod import f`	Ne charge en mémoire que la fonction `f` du module `mod`. On peut directement utiliser `f`.
`from mod import *`	Charge toutes les fonctions du module `mod`. On peut toutes les utiliser sous leur propre nom.

Voici une illustration dans un interpréteur Python :

```
>>> cos(0)
Traceback (most recent call
last):File "<console>", line 1,
in <module>NameError: name 'cos' is
not defined
>>> import numpy
>>> numpy.cos(0)
1.0
```

```
>>> cos(0)
Traceback (most recent call
last):File "<console>", line 1,
in <module>
NameError: name 'cos' is not defi-
ned
>>> type(cos)
<class 'numpy.ufunc'>
>>> numpy.cos(numpy.pi/2)
6.123031769111886e-17
```

9. Exercices

1 — *Quizz*

Qu'affichent les programmes suivants ? On répondra *sans les programmer*.

```
s,k=0,0
while s<24:
    k=k+1
    s=s+k
print(k)
```

```
s,k=0,0
while s<24:
    s=s+k
    k=k+1
print(k)
```

```
s,k=0,0
while k<24:
    s=s+k
    k=k+1
print(k)
```

```
s,k=0,0
while k<=24:
    k=k+1
    s=s+k
print(s)
```

2 — *Ma première fonction*

1. Écrire une fonction `max2(x,y)` qui renvoie le plus grand des deux nombres x et y.
2. En utilisant la fonction `max2`, écrire une fonction `max3` qui calcule le maximum de 3 nombres.

3 — *Ma première boucle*

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

4 — *La factorielle en version itérative*

Écrire une fonction `facto(n)` renvoyant 1 si $n = 0$ et $1 \times \cdots \times n$ si $n \in \mathbb{N}^*$. On donnera deux versions : avec boucle `for` puis boucle `while`.

5 — *La suite de Fibonacci f*

Écrire une fonction `Fibo` prenant en argument un entier n et renvoyant le terme d'indice n de la suite de Fibonacci définie par :

$$F_0 = 0 \quad , \quad F_1 = 1 \quad , \quad \forall n \in \mathbb{N},\ F_{n+2} := F_{n+1} + F_n$$

6 — *La conjecture de Syracuse ff*

La suite de Syracuse est définie par $u_0 \in \mathbb{N}^*$ et $\forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$

Par exemple, partant de $u_0 = 26$, on trouve successivement :

$$26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Plus généralement, on conjecture que, pour tout $u_0 \in \mathbb{N}^*$, il existe $n \in \mathbb{N}$ tel que $u_n = 1$.

1. Écrire une fonction `longueur(a)` d'argument un entier naturel a non nul renvoyant le plus petit entier naturel n tel que $u_n = 1$ où $(u_n)_{n\in\mathbb{N}}$ est la suite de Syracuse vérifiant $u_0 = a$.
2. Tester cette fonction pour les valeurs de a suivantes : 1000 et 10^5.
3. Écrire une fonction `maxVol(a)` d'argument un entier naturel a non nul renvoyant la plus grande valeur prise par la suite de Syracuse commençant en $u_0 = a$ avant d'atteindre la valeur 1 pour la première fois.
4. Tester cette fonction pour les valeurs de a suivantes : 1000 et 10^5.

7 — *Nombres premiers et conjectures* ***ff***

Le but de cet exercice est de déterminer les mille plus petits nombres premiers. Pour déterminer si un entier est premier, on utilise le critère suivant :

un entier p est premier si $p \geqslant 2$ et s'il n'est divisible par aucun entier $k \geqslant 2$ vérifiant $k^2 \leqslant p$

1. Écrire une fonction `estPremier`, prenant un paramètre entier p, et qui retourne le booléen `True` lorsque p est un nombre premier, et le booléen `False` dans le cas contraire.
2. Utiliser cette fonction pour afficher les mille plus petits nombres premiers.
3. La conjecture de Goldbach postule que tout entier pair supérieur à 3 peut s'écrire comme somme de deux nombres premiers (éventuellement égaux). Vérifier cette conjecture pour tout entier inférieur ou égal à 1000.
4. Montrer que la conjecture suivante est fausse : tout nombre impair est la somme d'une puissance de 2 et d'un nombre premier.

10. Indications

1

Aidez-vous d'un tableau afin de comprendre comment évoluent les différentes variables.

2

On peut créer une fonction `max3` ne contenant qu'une seule ligne de code.

3

L'adverbe *below* signifie <. Une boucle `for` s'impose et il faudra, en plus de la variable de la boucle, une variable `somme` qui contiendra le résultat escompté à la fin des itérations.

4

Attention à la boucle `while` : bien s'assurer du résultat au brouillon via (par exemple) un tableau d'évolution des variables.

5

En plus de la variable de la boucle `for`, utiliser deux variables `F0` et `F1` pour stocker deux termes consécutifs de la suite de Fibonacci.

6

Pour $a = 1000$ et $a = 10^5$, on trouve respectivement 111 et 128.

7

Au 1., on pourra itérer tant que $k^2 \leqslant n$ (pas besoin de racine carrée). Un nombre n est divisible par k non nul *si et seulement si* n modulo k est nul (utiliser l'opérateur %). Le millième nombre premier est 7919. Le plus petit entier impair pour lequel la conjecture du 4. est fausse est 127.