TP (S1) 1 Fondamentaux

Objectifs
- S'approprier l'environnement de travail.
- Revoir les éléments de base du langage : variables, tests, boucles, fonctions.

1. DÉCOUVREZ L'ENVIRONNEMENT DE TRAVAIL

1.1. Le réseau du lycée

Vous disposez d'un accès au réseau pédagogique du Lycée MONTAIGNE qui vous fournit un espace disque accessible depuis n'importe quel ordinateur du lycée.

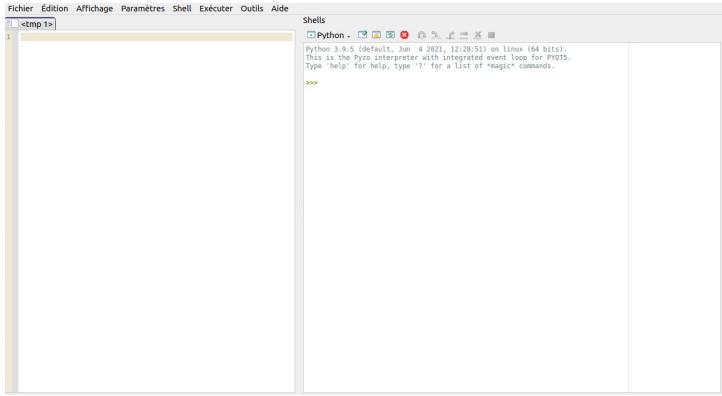
- Saisissez votre *identifiant* et votre *mot de passe* personnels pour vous connecter au réseau.
- Naviguez dans l'arborescence réseau et identifiez votre *dossier personnel* (dossier de travail). Vos fichiers doivent être enregistrés à cet emplacement, en créant au besoin des sous-dossiers (vous pouvez d'ores et déjà créer un dossier *ITC*, puis un sous-dossier *TP1*).
- Identifiez également le *dossier partagé*, qui est accessible à tous les élèves de la classe ainsi qu'aux enseignants. Vous y trouverez pour certaines séances des fichiers déposés par vos professeurs. Aujourd'hui vous devez voir apparaître un fichier nommé *tp1.py*. Copiez-le dans votre répertoire personnel (bien sûr dans le sous-répertoire *TP1*).

1.2. La distribution Pyzo

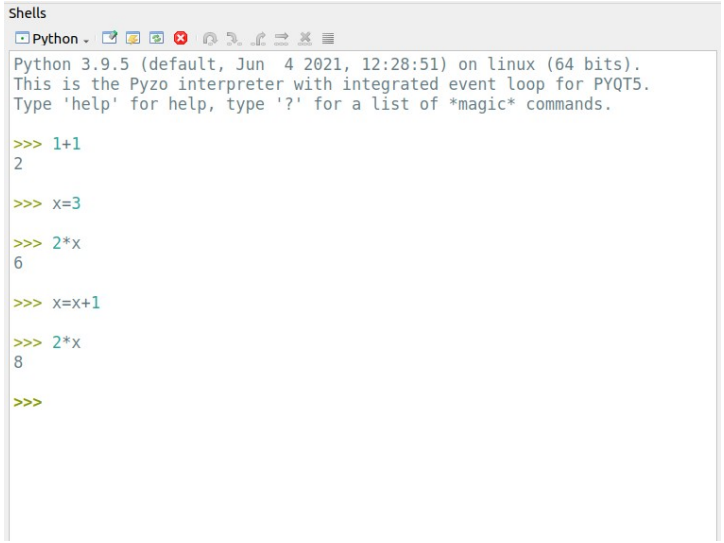
Pyzo est un environnement complet de travail associé au langage Python. Afin de prolonger le travail fait en classe, il est recommandé que vous l'installiez sur votre ordinateur personnel à partir du site http://www.pyzo.org (ce logiciel est bien sûr gratuit, et la procédure d'installation clairement détaillée).

> **Remarque 1** Il existe d'autres environnements de travail : IDLE (intégré par défaut à Python), ou encore Pycharm, Tony, *etc.*. On peut également directement faire appel à Python dans un terminal (ou une invite de commande sous Windows).

- Démarrez Pyzo.
- Après quelques instants, une fenêtre met à votre disposition différentes zones de travail. Deux zones sont particulièrement importantes pour vous : la zone d'*édition du code* et la *console* (ou shell) d'exécution des commandes, respectivement à gauche et à droite sur la copie d'écran suivante.

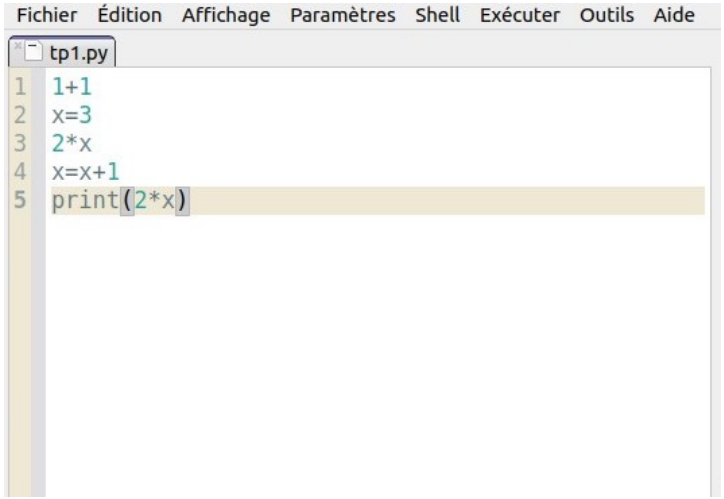


Les autres zones ne nous seront pas utiles, vous pouvez les fermer pour ne conserver que l'éditeur et la console.
- La *console* permet d'exécuter successivement une série d'instructions du langage, le résultat de chacune étant affiché en dessous de la commande.



Pour vous familiariser avec le fonctionnement de la console, tapez la succession d'instructions précédente.

- L'*éditeur* permet également de saisir une séquence d'instructions du langage, mais en sauvegardant celle-ci dans un *fichier* (vous pouvez choisir assez librement le nom des fichiers, mais ils ne doivent en particulier contenir ni espaces ni accents). Ouvrez dans l'éditeur le fichier *tp1.py* que vous avez copié dans votre répertoire personnel (menu *Fichier* puis *Ouvrir*).



Une différence importante est que l'ensemble des instructions du fichier sera exécuté, en une seule fois, et ce lorsque vous choisissez le menu *Exécuter* puis *Démarrer le script* (raccourci par la touche *F5*). Les résultats sont affichés dans la console, mais seulement ceux des instructions dont l'affichage est explicitement demandé par la commande `print`. En exécutant ce fichier vous ne devez donc voir que le résultat de la dernière instruction apparaître à l'écran :

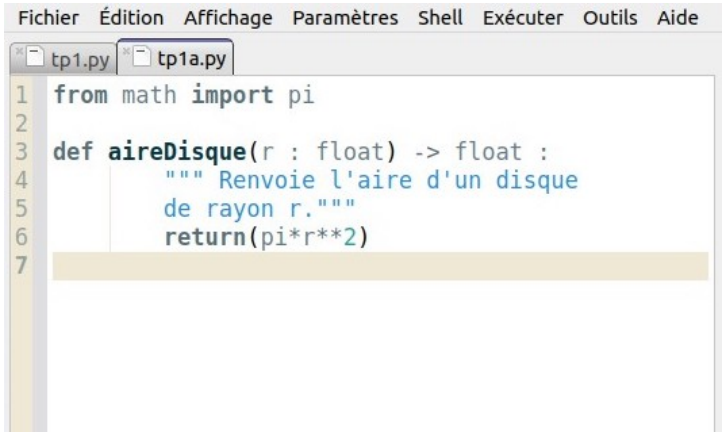
```
(executing file "tp1.py")
8
```

Ajoutez des instructions `print` sur d'autres lignes du fichier *tp1.py* et exécutez-le : vous verrez d'autres résultats intermédiaires apparaître. À l'inverse, si vous enlevez toutes les instruction `print` du fichier, les commandes seront bien exécutées mais sans aucun affichage.
Notez bien cette différence de comportement entre la console et l'éditeur : dans la console le résultat d'une instruction est systématiquement affiché sans qu'il soit nécessaire de recourir à la commande `print`, alors que dans l'éditeur seuls les affichages explicitement demandés par cette instruction sont effectués (les autres instructions sont bien sûr exécutées, mais sans que les résultats n'apparaissent à l'écran : il s'agit en pratique de résultats intermédiaires dont l'affichage ne ferait que remplir inutilement l'écran de la console ; nous verrons toutefois que demander temporairement certains affichages bien choisis facilite grandement la phase de « débogage » d'un programme). Notez aussi qu'il est possible de n'exécuter qu'une partie des instructions du fichier via le menu *Exécuter* puis *Exécuter la sélection*, après avoir sélectionné les lignes de code souhaitées.

2. DÉCLARER ET APPELER DES FONCTIONS

Notre activité de programmation consistera principalement à écrire des fonctions dans un fichier via l'éditeur.

Par exemple, ouvrez dans l'éditeur une nouvelle page, sauvegardez-la dans votre répertoire *TP1*, par exemple sous le nom *TP1a.py*, et saisissez la déclaration de fonction suivante :



Pour utiliser cette fonction vous avez deux possibilités :

- soit exécuter le fichier contenant uniquement la déclaration de la fonction (vous ne verrez aucun résultat dans la console à cette étape) :

  ```
  (executing file "tp1a.py")
  ```

  puis, toujours dans la console, appeler la fonction sur une valeur (dans ce cas, vous aurez compris qu'il n'est pas nécessaire d'utiliser l'instruction `print`) :

  ```
  aireDisque(1.23)
  4.752915525615998
  ```

- soit appeler la fonction directement dans le fichier, en créant un *corps principal* au programme, à la suite de la déclaration de la fonction et sans indentation, pour que cette instruction ne fasse plus partie de la fonction (dans ce cas bien sûr, il est nécessaire d'utiliser l'instruction *print* pour que le résultat soit affiché) :

Le résultat dans la console sera :

```
(executing file "tp1a.py")
4.752915525615998
```

Enfin, notons qu'il est bien sûr possible d'utiliser dans le corps d'une fonction une autre fonction définie au préalable. Par exemple l'exécution du programme suivant entré dans l'éditeur :

```
def f(x : float) -> float :
    return(x+1)

def g(x : float) -> float :
    return(x**2)

def h(x : float) -> float :
    return(g(f(x)))
```

puis dans la console :

```
>>> h(2)
9
```

En effet, l'évaluation de `h(2)` provoque le calcul de `g(f(2))`, donc en premier l'appel `f(2)` qui renvoie 3, puis l'appel `g(3)` qui renvoie bien 9.

3. BOUCLES

3.1. Boucle `for`

Exercice 1 **Calculer les termes d'une suite récurrente** [Sol 1]

1) On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = 1 \\ \forall\, n \in \mathbb{N}, \quad u_{n+1} = 2u_n + 3. \end{cases}$$

1.1) Précisez les valeurs de u_n pour $n \in \{1,2,3,4,5\}$.

1.2) Le calcul de u_n peut bien sûr se faire à l'aide de la fonction suivante :

```
def suiteU(n : int) -> int:
    """ Calcule u_n """
    u = 1
    for k in range(1,n+1):        k pouvant être remplacé ici par un underscore
        u = 2*u+3
    return(u)
```

Pour s'en convaincre, on peut ranger dans un tableau les valeurs successives prises par la variable u lors de l'appel `suiteU(5)`, la notation u_i désignant le contenu de u à la fin du i^e passage dans la boucle `for` (par convention, u_0 désigne le contenu de u juste avant de rentrer pour la première fois dans la boucle) :

i	u_i
0	1 (car u est initialisé à 1 en ligne #3)
1	5 (car $u_1 = 2u_0 + 3 = 2 \times 1 + 3 = 5$)
2	13
3	29
4	61
5	125

Cette valeur 125 contenue dans la variable u à la sortie de la boucle `for` est celle renvoyée par l'appel `suiteU(5)`, et on constate que c'est bien la valeur de u_5. Tapez cette fonction dans l'éditeur (en la sauvegardant dans un fichier du nom de votre choix) et utilisez-la pour calculer la valeur de u_{100}.

2) On considère maintenant la célèbre suite de SYRACUSE définie par $u_0 = a$, où a est un entier naturel non nul à préciser, et pour tout $n \in \mathbb{N}$:

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

2.1) Précisez les valeurs de u_n pour $n \in \{1,2,3,4,5\}$ dans le cas où $a = 6$.

2.2) On rappelle que les commandes % et // renvoient respectivement le reste et le quotient de la division euclidienne de deux entiers. Par exemple `25%7` et `25//7` renvoient respectivement 4 et 3 puisque :

$$25 = 3 \times 7 + 4 \quad \text{avec} \quad 0 \leqslant 4 < 7.$$

Pour calculer u_n on propose la fonction suivante, où les points de suspension sont à compléter :

```
def suiteU(n : int, a : int) -> int :
    """ Calcule u_n """
    u = ...
    for k in range(1, n+1) :
        if u%2 == ... :
            u = u//2
        else :
            u = ...
    return(u)
```

Tapez le code en le complétant, et testez-le pour retrouver la valeur de u_5 pour $a = 6$.

Exercice 2 Sommer/multiplier et compter [Sol 2]

1) 1.1) On considère la fonction suivante :

```
def Devine(N : int) -> int :
    """ A compléter """
    S = 0
    for k in range(1, N+1) :
        S += k
    return(S)
```

i) Déterminer la valeur renvoyée par l'appel de `Devine(5)` en complétant le tableau suivant (dont les trois premières lignes vous sont données) qui détaille les valeurs successives prises par les variables k et S au fur et à mesure des passages dans la boucle for :

i	k_i	S_i
0	n'existe pas encore	0
1	1	1
2	2	3
3		
4		
5		

ii) Quel est le sens à donner à la valeur renvoyée par l'appel `Devine(N)` ? Modifiez l'en-tête de la fonction pour préciser ce qu'elle renvoie.

1.2) Inspirez-vous de la fonction précédente pour écrire une fonction `factorielle(N : int) -> int` où N est un entier naturel, et renvoyant sa factorielle définie par : $\begin{cases} N! = 1 \times 2 \times \cdots \times N & \text{si } N > 0, \\ 0! = 1. \end{cases}$ *Vérifiez bien que votre fonction est correcte dans le cas où* $N = 0$.

2) 2.1) On souhaite compter combien il y a de nombres premiers entre 1 et un entier naturel non nul N donné. Par exemple, pour $N = 5$ on en trouve 3 (qui sont 2, 3 et 5). Pour cela, on considère la fonction suivante :

```
from sympy import isprime # cette fonction permet  de \
↪ tester si un entier est un nombre premier

def nbPrem(N : int) -> int :
    """ Compte le nombre d'entier de 1 à N qui sont
    des nombres premiers """
    nb = 0
    for k in range(1, N+1):
        if isprime(k):
            nb += 1
    return(nb)
```

Complétez le tableau suivant pour l'appel de `nbPrem(5)` :

i	k_i	$\texttt{nb}_i$
0		
1		
2		
3		
4		
5		

2.2) Inspirez-vous de la fonction précédente pour écrire une fonction `nbSinPos(N : int) -> int` où N est un entier naturel non nul, renvoyant le nombre d'entiers k dans $\{1, \ldots, N\}$ tels que $\sin(k) \geqslant 0$. *(La fonction `sin` est à importer du module `math`.)*

Exercice 3 **Double boucle** [Sol 3] Écrire une fonction `nbCouples(N : int) -> int` où N est un entier naturel, et retournant le nombres de couples d'entiers (x, y) tels que $0 \leqslant x \leqslant 5, 0 \leqslant y \leqslant 5$ et $x^2 + y^2 \leqslant N$. Testez sur plusieurs valeurs de N, que doit-on observer lorsque N grandit ?

Exercice 4 **Des étoiles plein les yeux** [Sol 4] Écrire des fonctions prenant en argument le nombre de lignes `n` (ici `n = 5`) et permettant l'affichage des figures suivantes.

1.
```
★ ★ ★ ★ ★
★ ★ ★ ★ ★
★ ★ ★ ★ ★
★ ★ ★ ★ ★
★ ★ ★ ★ ★
```

2.
```
    ★
   ★ ★
  ★ ★ ★
 ★ ★ ★ ★
★ ★ ★ ★ ★
```

3.
```
★ ★ ★ ★ ★
 ★ ★ ★ ★
  ★ ★ ★
   ★ ★
    ★
```

4.
```
★ ★ ★ ★ ★
 ★ ★ ★ ★
  ★ ★ ★
   ★ ★
    ★
```

Indication : On pourra commencer par observer par exemple le résultat de l'instruction `"*"*3+""*2` dans la console

Exercice 5 **Calculer un maximum** [Sol 5] On considère dans cet exercice que l'on dispose d'une fonction `f(n : int) -> float`.

1) On souhaite écrire une fonction `Maximum(N : int) -> float` où N est un entier naturel, et renvoyant la plus grande des valeurs $f(0), \ldots, f(N)$.
Pour cela on propose le code suivant (qui comporte une erreur comme nous allons le constater) :

```
def Maximum(N : int) -> float :
    """ Calcule le maximum de f(0),...,f(N) """
    M = f(0)
    for k in range(1, N+1) :
        if f(k) > f(k-1) :
            M = f(k)
    return(M)
```

1.1) On suppose que la fonction f est telle que

$$f(0) = 5, \quad f(1) = 2, \quad f(2) = 3, \quad f(3) = 7, \quad f(4) = 1, \quad f(5) = 4.$$

Détailler dans un tableau l'appel de `Maximum(5)` pour déterminer quelle sera la valeur renvoyée, et constatez qu'elle n'est pas correcte.

1.2) Modifier le code de la fonction pour qu'elle soit correcte, et utiliser cette fonction pour déterminer la plus grande valeur de $\sin(k)$ pour $0 \leqslant k \leqslant 100$ (il vous faudra bien sûr définir la fonction f correspondante).

1.3) En vous inspirant de la fonction précédente, écrire une fonction `MaximumAtteintEn(N : int) -> int` renvoyant un entier $0 \leqslant k \leqslant N$ tel que $f(k)$ soit la valeur maximale parmi $f(0), \dots, f(N)$. *(La fonction ne devra pour cela n'employer qu'une seule boucle* **`for`***. D'autre part, il peut arriver que plusieurs valeurs de k permettent d'atteindre de le maximum, auquel cas la fonction renverra la plus petite.)*
Utilisez cette fonction pour déterminer k tel que $\sin(k)$ soit le maximum de $\sin(0), \dots, \sin(100)$.

2) Modifier les fonctions précédentes pour écrire les fonctions `Minimum(N : int) -> float` et `MinimumAtteintEn(N : int) -> int` correspondantes.

Exercice 6 **Les nombres parfaits** [Sol 6] On dit qu'un entier naturel non nul n est un nombre *parfait* s'il est égal au double de la somme de ses diviseurs positifs.

- Par exemple 6 est un nombre parfait car les diviseurs positifs de 6 sont $1, 2, 3, 6$ et $1+2+3+6 = 12 = 2 \times 6$.
- Par contre 8 n'est pas un nombre parfait car les diviseurs positifs de 8 sont $1, 2, 4, 8$ mais : $1+2+4+8 = 15 \neq 2 \times 8$.

Écrire une fonction `estParfait(n : int) -> bool` où n est un entier naturel non nul, et renvoyant le booléen **`True`** si n est un nombre parfait et **`False`** sinon. Utiliser cette fonction pour faire afficher à l'écran les nombres parfaits compris entre 1 et 10000.

3.2. Boucle while

Exercice 7 **Suite récurrente d'ordre 1** [Sol 7] Si a est un réel strictement positif, on peut démontrer que la suite définie par :

$$\begin{cases} u_0 = a \\ \forall\, n \geqslant 0,\ u_{n+1} = \frac{u_n^2 + a}{2u_n} \end{cases}$$ converge vers $\sqrt{a}$.

1) Écrire une fonction `suiteU(a : float, n : int) -> float` où n est un entier naturel, et renvoyant u_n. Calculer u_4 pour $a = 2$, et comparer à une valeur approchée de $\sqrt{2}$.

2) On souhaite maintenant écrire une fonction `approxRacCarrée(a : float, e : float) -> float` où $a \in \mathbb{R}^+$ et $e \in \mathbb{R}^{+*}$, calculant le premier terme de la suite vérifiant $|\sqrt{a} - u_n| \leqslant e$ (ceci signifie que u_n est une valeur approchée de $\sqrt{a}$ à la précision e).

Cela revient principalement à remplacer la boucle **`for`** par une boucle **`while`** dans la fonction précédente :

```
from math import sqrt   # sqrt calcule la racine carrée
def approxRacCarrée(a : float, e : float) -> float :
    """ Renvoie le premier u_n tel que |sqrt(a)-u\_n| soit
    inférieur à e """
    u = a
    while abs(sqrt(a)-u) > e : # abs : valeur absolue
        u = (u**2+a)/(2*u)
    return(u)
```

Noter que la condition écrite dans la boucle **`while`** : `abs(sqrt(a)-u) > e` est celle pour **rester** dans la boucle, et donc c'est bien **négation** de la condition

$$|\sqrt{a} - u_n| \leqslant e$$

que l'on veut voir respectée au **sortir** de la boucle. Taper cette fonction, et la tester en obtenant une valeur approchée de $\sqrt{7}$ à 10^{-10} près.

Exercice 8 [Sol 8] On considère la fonction suivante :

```
from sympy import isprime # Teste si un entier est premier
def Devine(n : int) -> int :
    k = n
    while not(isprime(k)):
        k += 1
    return(k)
```

1) Dérouler à la main l'appel `Devine(8)` (on fera figurer dans un tableau les valeurs successives prises par la variable `k` après chaque passage dans la boucle **`while`**).

2) Quel est le sens à donner au résultat de l'appel `Devine(n)` ? Modifier l'en-tête de cette fonction pour expliciter l'usage.

Exercice 9 **SYRACUSE, le retour** [Sol 9] On revient dans cette question sur la suite de SYRACUSE définie par $u_0 = a$, où a est un entier naturel non nul, et pour tout $n \in \mathbb{N}$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Une célèbre conjecture est que cette suite finit toujours par atteindre la valeur 1, et ce quelque soit son premier terme a choisi, puis bien sûr boucler sur les valeurs $1, 4, 2, 1, 4, 2, \ldots$.
Par exemple pour $a = 6$ les premières valeurs de la suite sont :

$$u_0 = 6, u_1 = 3, u_2 = 10, u_3 = 5, u_4 = 16, u_5 = 8, u_6 = 4, u_7 = 2, \boxed{u_8 = 1}, \ldots$$

Écrire une fonction `SyracusePremUn(a : int) -> int` où a est un entier naturel non nul, et renvoyant le plus petit entier naturel n tel que $u_n = 1$.
La tester pour $a = 127$.

Exercice 10 **Algorithme d'EUCLIDE** [Sol 10] Si a et b sont deux entiers naturels, on rappelle que pgcd(a, b) (plus grand commun diviseur de a et b) peut se calculer par l'algorithme d'EUCLIDE :

- si $b = 0$ alors pgcd$(a, b) = a$;
- sinon, on calcule le reste r de la division euclidienne de a par b. Or on peut démontrer que pgcd(a, b) = pgcd(b, r), donc on remplace la valeur de a par celle de b, et la valeur de b par celle de r.

On itère le procédé jusqu'à trouver $b = 0$. La valeur de pgcd(a, b) est alors contenue dans a.

1) On prend $a = 192$ et $b = 138$. Faites fonctionner « à la main » l'algorithme d'EUCLIDE en complétant le tableau suivant *(la première étape est déjà remplie, 54 étant le reste de la division euclidienne de 192 par 138)* :

a	b	r
192	138	54
138	54	…
…	…	…
…	…	…
…	…	…
…	…	…

Préciser la valeur de pgcd$(192, 138)$.

2) On souhaite écrire une fonction `pgcd(a : int, b : int) -> int` où a et b sont deux entiers naturels, et renvoyant `pgcd(a,b)`.
Le code suivant, à compléter, vous est donné :

```
def pgcd(a : int, b : int) -> int :
    """ Renvoie le pgcd de a et b """
    while ... :
        r = a%b
        a,b = ...
    return(a)
```

Utiliser cette fonction pour calculer pgcd$(51940, 6201)$.

Exercice 11 [Sol 11] On se donne un entier naturel n. Si n s'écrit avec plusieurs chiffres, alors on calcule le produit de ses chiffres ce qui nous donne un nouvel entier naturel. On recommence ce procédé sur le nombre obtenu, jusqu'à avoir un nombre à un seul chiffre.

Par exemple en partant de $n = 377$, on trouve successivement :

- $3 \times 7 \times 7 = 147$ à la première étape ;
- $1 \times 4 \times 7 = 28$ à la deuxième étape ;
- $2 \times 8 = 16$ à la troisième étape ;
- $1 \times 6 = 6$ à la quatrième et dernière étape étape.

On appelle *persistance* de l'entier n le nombre d'étapes qui ont été nécessaires pour le réduire à un seul chiffre. Sur notre exemple, la persistance de 377 vaut donc 4. Notez que si n est déjà formé d'un seul chiffre, alors sa persistance vaut 0.

1) Écrire une fonction `prodChiffres(n : int) -> int` où n est un entier naturel, et renvoyant le produit des chiffres de n.
(On remarquera que `n%10` donne le chiffre des unités de n, et `n//10` le nombre obtenu en enlevant ce dernier chiffre.)

2) Écrire une fonction `persistance(n : int) -> int` où n est un entier naturel, et renvoie sa persistance.
Calculer la persistance de 277777788888899.

3) Une question légitime consiste à rechercher des entiers naturels ayant une persistance élevée. Or une conjecture célèbre émet l'hypothèse que la persistance maximale que l'on peut obtenir est seulement égale à 11.
Écrire une fonction `minPersistance(p : int) -> int` où p est un entier naturel, renvoyant le plus petit entier naturel n dont la persistance est égale à p.
Par exemple, `minPersistance(3)` renvoie 39, car on peut vérifier que 39 est de

persistance 3 et que tous les entiers de 0 à 38 ont une persistance inférieure ou égale à 2.
Utiliser la fonction pour calculer `minPersistance(7)`.
(Notez que les calculs deviennent bien sûr de plus en plus long quand p augmente, et vous aurez sans doute du mal à dépasser $p = 8$ en un temps raisonnable.)

Exercice 12 **Des nombres bien rangés** [Sol 12] Pour finir en beauté, cet exercice nettement plus difficile s'adresse à celles et ceux d'entre-vous les plus à l'aise en programmation. L'énoncé est extrait des phases qualificatives 2017 du concours de programmation **Google Code Jam** (*https://codingcompetitions.withgoogle.com/*) et vous fera en outre travailler votre anglais :

Tatiana likes to keep things tidy. Her toys are sorted from smallest to largest, her pencils are sorted from shortest to longest and her computers from oldest to newest. One day, when practicing her counting skills, she noticed that some integers, when written in base 10 with no leading zeroes, have their digits sorted in non-decreasing order. Some examples of this are 8, 123, 555, and 224488. She decided to call these numbers tidy. Numbers that do not have this property, like 20, 321, 495 and 999990, are not tidy. She just finished counting all positive integers in ascending order from 1 to N. What was the last tidy number she counted?

Comme vous l'aurez compris, un entier d'écriture décimale $c_n c_{n-1} \dots c_2 c_1$ est dit « bien rangé » (ie. tidy) si : $c_n \leqslant c_{n-1} \leqslant \dots \leqslant c_2 \leqslant c_1$.
Par exemple l'entier 3356889 est bien rangé, mais 3354889 ne l'est pas.
L'objectif est donc d'écrire une fonction `tidy(N : int) -> int` où N est un entier naturel non nul, et renvoyant le plus grand entier bien rangé inférieur ou égal à N. Ajoutons que les fonctions à écrire pour le concours Google Code Jam doivent obéir à un soucis d'efficacité en fournissant leurs réponses dans un temps « raisonnable ». Par exemple, le résultat de l'appel `tidy(11222000001222556799)` devra être obtenu « instantanément » (en moins d'une seconde). Enfin, nous n'utiliserons aucune structure de données complexe (listes, chaînes de caractères, dictionnaires, …), ce qui aurait pu être approprié mais fera l'objet de TP ultérieurs.

SOLUTIONS DES EXERCICES

Solution 1

1) 1.1) $u_1 = 5$, $u_2 = 13$, $u_3 = 29$, $u_4 = 61$, $u_5 = 125$.

1.2) L'appel de `suiteU(100)` (dans l'éditeur avec `print` ou dans la console sans `print`) donne $u_{100} = 507060240091291760598681282150 1$.

2) 2.1) $u_1 = 3$, $u_2 = 10$, $u_3 = 5$, $u_4 = 16$, $u_5 = 8$.

2.2)

```
def suiteU(n : int, a : int) -> int :
    """ Calcule u_n """
    u = a
    for k in range(1,n+1) :
        if u%2 == 0 :
            u = u//2
        else :
            u = 3*u+1
    return(u)
```

Solution 2

1) 1.1) i)

i	k_i	S_i
0	n'existe pas encore	0
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15

ii)

```
def Somme(N : int) -> int :
    """ Renvoie 1+2+...+N """
```

1.2)

```
def Factorielle(N : int) -> int :
    """ Renvoie 1x2x...xN si N>0 et 1 si N=0"""
    P = 1
    for k in range(1, N+1) :
        P *= k
    return(P)
```

2) 2.1)

i	k_i	nb_i
0	n'existe pas	0
1	1	0
2	2	1
3	3	2
4	4	2
5	5	3

2.2)

```
from math import sin

def nbSinPos(N : int) -> int :
    """ Compte le nombre d'entier k de 1 à N tels que
    sin(k) soit positif """
    nb = 0
    for k in range(1, N+1) :
        if sin(k) >= 0 :
            nb += 1
    return(nb)
```

Solution 3

```
def nbCouples(N : int) -> int:
    nb = 0
    for x in range(0, 6):
        for y in range(0, 6):
            if x**2+y**2 <= N:
                nb += 1
    return nb
```

```
>>> nbCouples(1)
3
>>> nbCouples(10)
13
>>> nbCouples(20)
22
>>> nbCouples(50)
36
>>> nbCouples(60)
36
```

Solution 4

```
def fig1(n):
    for k in range(n):
        l = ''
        for i in range(n):
            l = l+'*'
        print(l)


def fig2(n):
    for k in range(n):
        l = ''
        for i in range(k+1):
            l = l+'*'
        print(l)

def fig3(n):
    for k in range(n):
        l = ''
        for i in range(n-k):
            l = l+'*'
        print(l)

def fig4(n):
    for k in range(n):
        l = k*' ' + (n-k)*'*'
        print(l)
```

```
>>> fig1(5)
*****
*****
*****
*****
*****
>>> fig2(5)
*
**
***
****
*****
>>> fig3(5)
*****
****
***
**
*
>>> fig4(5)
*****
 ****
  ***
   **
    *
```

Solution 5

1) 1.1)

i	k_i	M_i
0	n'existe pas	5
1	1	5
2	2	3
3	3	7
4	4	7
5	5	4

donc l'appel renvoie 4, alors que le maximum est 7.

1.2)

```python
from math import sin

def f(n : int) -> float :
    """ Renvoie sin(n) """
    return(sin(n))

def Maximum(N : int) -> float :
    """ Calcule le maximum de f(0),...,f(N) """
    M = f(0)
    for k in range(1,N+1) :
        if f(k)>M :
            M = f(k)
    return(M)
```

```
>>> Maximum(100)
0.9999118601072672
```

1.3)

```python
from math import sin

def f(n : int) -> float :
    """ Renvoie sin(n) """
    return(sin(n))

def MaximumAtteintEn(N : int) -> int :
    """ Renvoie k compris entre 0 et N tel que
    f(k) soit le maximum de f(0),...,f(N) """
    indM = 0
    for k in range(1, N+1) :
        if f(k) > f(indM) :
            indM = k
    return(indM)
```

```
>>> MaximumAtteintEn(100)
33
```

2)

```python
def Minimum(N : int) -> float :
    """ Calcule le minimum de f(0),...,f(N) """
    M = f(0)
    for k in range(1,N+1) :
        if f(k) < M :
            M = f(k)
    return(M)
```

```python
def MinimumAtteintEn(N : int) -> int :
    """ Renvoie k compris entre 0 et N tel que
    f(k) soit le minimum de f(0),...,f(N) """
    indM = 0
    for k in range(1, N+1) :
        if f(k) < f(indM) :
            indM = k
    return(indM)
```

```
>>> Minimum(100)
-0.9999902065507035
>>> MinimumAtteintEn(100)
11
```

Solution 6

```python
def estParfait(n : int) -> bool :
    """ Renvoie True si n est parfait et False sinon """
    S = 0
    for k in range(1,n+1) :
        if n%k==0 : # teste si k divise n
            S += k
    return(S==2*n)
```

```
>>> estParfait(6)
True
>>> estParfait(8)
False
>>> for n in range(1, 10001) :
...     if estParfait(n) :
...         print(n)
...
6
28
496
8128
```

Solution 7

1)
```
from math import sqrt   # sqrt calcule la racine carrée

def suiteU(a : float, n : int) -> float :
    """ Renvoie u_n """
    u = a
    for k in range(1, n+1) :
        u = (u**2+a)/(2*u)
    return(u)
```

```
>>> suiteU(2,4)
1.4142135623746899
>>> sqrt(2)
1.4142135623730951
```

2)
```
>>> approxRacCarrée(7,10**(-10))
2.6457513111113693
>>> sqrt(7)
2.6457513110645907
```

Solution 8

i	k_i
0	8 (non premier)
1	9 (non premier)
2	10 (non premier)
3	11 (premier)

1) L'appel `Devine(8)` renvoie donc 11.

2)
```
def nombrePremierSuivant(n : int) -> int :
    """ Renvoie le plus petit nombre premier
    supérieur ou égal à n """
```

Solution 9

```
def SyracusePremUn(a : int) -> int :
    """ Calcule le premier indice où
    la suite de \textsc{Syracuse} vaut 1 """
    u = a
    n = 0
    while u != 1:
        if u%2 == 0 :
            u = u//2
        else :
            u = 3*u+1
        n += 1
    return(n)
```

```
>>> SyracusePremUn(127)
46
```

Solution 10

1) $\text{pgcd}(192, 138) = 6$:

a	b	r
192	138	54
138	54	30
54	30	24
30	24	6
24	6	0
6	0	

2)
```
def pgcd(a : int, b : int) -> int :
    """ Renvoie le pgcd de a et b """
    while b!=0 :
        r = a%b
        a,b = b,r
    return(a)
```

```
>>> pgcd(192,138)
6
```

Solution 11

1)
```
def prodChiffres(n : int) -> int :
    """ Renvoie le produit des chiffres
    de n """
    P = n%10
    n = n//10
```

```
    while n != 0 :
        P = P*(n%10)
        n = n//10
    return(P)
```

```
>>> prodChiffres(123)
6
```

2)
```
def persistance(n : int) -> int :
    """ Renvoie la persistance de n """
    k = 0
    while n > 9 :
        n = prodChiffres(n)
        k += 1
    return(k)
```

```
>>> persistance(277777788888899)
11
```

3)
```
def minPersistance(p : int) -> int :
    """ Renvoie le plus petit entier
        de persistance égale à n """
    n = 0
    while persistance(n) != p :
        n += 1
    return(n)
```

```
>>> minPersistance(7)
68889
```

Solution 12 Une première idée serait d'écrire une fonction testant si un entier est bien rangé, puis de tester à l'aide d'une boucle **while** successivement les entiers à partir de N et dans l'ordre décroissant jusqu'à trouver le premier bien rangé. Le problème de cette approche est que, si N est grand et que le plus grand bien rangé est très éloigné de N, le temps de calcul est généralement prohibitif car il y aura un très grand nombre d'entiers à tester.

On cherche alors une approche plus fine, consistant à comprendre comment se construit l'entier cherché. Prenons l'exemple de :

N = 11222200001222556799.

L'idée est de repérer le premier (de la gauche vers la droite) chiffre mal rangé :

N = 112222$\boxed{0}$0001222556799

puis repérer une succession éventuelle d'un même chiffre à sa gauche :

N = 11$\underline{2222}$00001222556799.

Pour obtenir l'entier bien rangé cherché, il suffit de diminuer de un le premier chiffre de cette succession, et remplacer tous les chiffres suivants de N par 9 :

`tidy(N)` = 11199999999999999999.

```
def nbChiffres(N : int) -> int :
    """ Renvoie le nombre de chiffres de N """
    nbc = 0
    while N > 0 :
        nbc += 1
        N = N//10
    return(nbc)

def kemeChiffre(N : int, k : int) -> int :
    """ Renvoie le k-ième chiffre de N
    (en partant de la droite) """
    for i in range(k-1) :
        N = N//10
    return(N%10)

def tidy(N : int) -> int :
    """ Renvoie le plus grand nombre bien rangé
        inférieur ou égal à N """

    # Détermination si N est bien rangé et, si non, du
    # premier chiffre mal rangé et du début d'une succession
    # d'un même chiffre à sa gauche (de rang p)
    nbc = nbChiffres(N)
    i,p = nbc,nbc
    chiffreCourant = kemeChiffre(N, i)
    bienRange = True
    while bienRange and i > 1 :
        chiffrePrecedent = kemeChiffre(N, i-1)
        if chiffreCourant <= chiffrePrecedent :
            if chiffreCourant != chiffrePrecedent :
                p = i-1
            i = i-1
```

```
            chiffreCourant = chiffrePrecedent
        else :
            bienRange = False

    # Si N est bien rangé, c'est l'entier cherché
    if bienRange:
        return(N)
    # Sinon, on construit l'entier
    else :
        R = 0
        for i in range(nbc, 0, -1) :
            R = R*10
            if i > p :
                R += kemeChiffre(N,i)
            elif i == p :
                R += kemeChiffre(N,i)-1
            else :
                R += 9
        return(R)
```

```
>>> tidy(11222200001222556799)
11199999999999999999
```