Informatique - TD 1 | Retour sur les basiques

Ce document nécessite d'ouvrir le fichier "TP1 eleves 2024-25.py" qui est sur le site bcpst2.f1.free.fr. À défaut, il vous faudra recopier les programmes proposés.
Pour ce faire, rendez-vous à la partie "Informatique" puis "TP". Faites un clic droit sur le carré dans la case "document" du TP1 et enregistrez le fichier lié. Ouvrez le pour finir (depuis Pyzo de préférence). ***Attention !*** *Si vous cliquez sur le carré et faites un copié-collé, vous risquez de gros problèmes d'affichage et d'éxecution...*

Script et fonctions

Supposons $a \geqslant 0$ et $n \in \mathbb{N}$. L'algorithme ci-dessous a été écrit pour afficher le résultat de la valeur S_n telle que

$$S_n = \begin{cases} \sum_{k=1}^{n} \sqrt{1+ak} & \text{si } n \in \mathbb{N}^* \\ 0 & \text{si } n = 0. \end{cases}$$

```
1  n=5
2  a=3
3
4  import math as mt
5  s=0
6  i=0
7  while i<n:
8  i+=1
9  s+=mt.sqrt(1+a*i)
10 print('Pour n='+str(n),'le résultat est',str(s) )
```

Exercice 1: [Correction] L'indentation

Indentez ce programme correctement de manière à ce qu'il donne a priori le résultat souhaité. **Attention**, on ne demande pas d'intégrer ceci dans une fonction à ce stade !
(une fois terminé, il doit fonctionner si on le fait tourner ! Il doit donner environ `15.41...`.)

Exercice 2: [Correction] inuput jusqu'à obtention d'un nombre correct

On souhaite maintenant faire rentrer les $''a''$ et le $''n''$ par l'utilisateur à l'aide d'un input et lui redemander, s'il y a erreur sur les conditions, jusqu'à avoir un $''a''$ et un $''n''$ correct. On propose d'utiliser les lignes suivantes pour les demandes :

```
1  a=float(input('Donner un réel a>=0 : '))
2  n=float(input('Donner un nombre entier n : '))
```

En effet, on se souvendra que la fonction `input` rend un texte. Il faut donc transformer le résultat en un réel (d'où le "`float`"). On pourra également utiliser la fonction `print` afin d'afficher des messages supplémentaires si besoin.
En retirant les deux lignes ci-dessus de numéro 1 et 2 (i.e. les commandes `n=5` et `a=3`), rajouter maintenant dans le script les lignes nécessaires à ce qui est demandé. (On pensera à utiliser une boucle à l'aide d'un `while` jusqu'à obtention de "bonnes" valeurs.)

Exercice 3: [Correction] Le mettre dans une fonction

Même si l'agorithme précédent tourne parfaitement bien de cette façon, on sait qu'il peut être utile, dans de nombreux cas, d'en disposer sous forme de fonction. En effet, on peut avoir à y faire appel plusieurs fois, ou encore de l'utiliser au sein d'un autre algorithme sans tout recopier (ou faire de copié collé...)

En retirant la partie contenant les `input`, transformer l'agorithme en une fonction `somme` qui prend comme entrée a, n et qui retourne :
- la valeur de S_n si $n \in \mathbb{N}$ et que $a \geqslant 0$.
- **None** sinon.

(On notera qu'on on ne demande plus cette fois-ci de redemander jusqu'à obtention des "bons a, n".)
Attention*, on demande que la fonction retourne une valeur (type "float") et non une phrase ou tout autre type. On fera donc attention au* `print` *qui se trouve dans la ligne initiale 10, et qui n'est plus du tout adapté. On doit par exemple pouvoir demander* `somme(2,3)/6` *(i.e. $a = 2$ et $n = 3$) sans avoir de message d'erreur.*

Commentaires et structure du document

Exercice 4: [Correction] Les commentaires

On rappelle que, dans le cadre de la construction d'une fonction, nous avons à notre disposition deux types de commentaires, : ceux écrit en `'''...'''` ou `"""..."""` et ceux écrit après un `#`.

Les deux premiers (`'''...'''` ou `"""..."""`) sont ceux qui servent à décrire la fonction. **De ce fait, il n'a d'utilité pour vous que dans le cadre des fonctions !** C'est ce qui s'affiche si on on tape ensuite `help(`*nom de la fonction*`)` dans le

shell. De manière générale, au minimum, on obtient généralement les informations suivantes : *le nombre d'arguments, leur type, ce que rend la fonction.*
Essayez avec une fonction de votre choix. *(Si vous n'êtes pas inspiré(e)s, je vous propose la fonction* `round`*.*
Observez au passage qu'elle peut avoir un ou deux arguments au choix selon ce que vous souhaitez. On peut ainsi taper `round(1.265)` *qui rend* `1` *ou encore* `round(1.265,1)` *qui rend* `1.3`*. Ceci est lié à une possibilité de rentrer des arguments dits "par défaut" dans une fonction. Nous verrons cela un peu plus loin.)*

Le dernier (#) est celui qu'on ne peux voir que si on s'intéresse au code exact de l'algorithme. **On peut s'en servir à l'intérieur une fonction ou n'importe où ailleurs dans votre script !**
Il sert à expliquer les variables introduites et à décrire les actions effectuées dans le script. Dans le cas de certains éditeurs de texte, il sert également à découper le script en "cellules" qui peuvent être compilées separément. J'y reviendrai dans l'exercice suivant.

1. Complétez votre fonction `somme` de manière à obtenir l'explication de ce que fait la fonction si on tape `help(somme)`.
2. Commentez l'algorithme pour le rendre plus lisible et compréhensible par un lecteur.
3. Appelez l'enseignant pour vérification.

Exercice 5: [Correction] Les cellules

On rappelle (ou non...) que dans Pyzo et Spyder, il est possible de découper son code en plusieurs sous-parties qui peuvent être éxécutées séparément éventuellement par des raccourcis clavier pour être plus rapide. On propose de considérer les quelques lignes suivantes sous Pyzo :

```
## Cellule 1
print("Vous avez bien compilé la cellule 1.")

## Cellule 2
print("C'est ici la cellule 2 qui a été compilée.")

##
```

ou alors sous Spyder :

```
# %% Cellule 1
print("Vous avez bien compilé la cellule 1.")

# %% Cellule 2
print("C'est ici la cellule 2 qui a été compilée.")

# %%
```

1. Pour exécuter la cellule 1 : Placez votre curseur à n'importe quel endroit dans la cellule "1". Ici, par exemple sur la ligne 2 ou 3.

   Sous Pyzo : Allez dans le menu `Run` puis `Execute Cell`. Cliquez dessus et profitez en pour regardez le raccourci clavier qui vous est proposé. Il change un peu suivant votre système d'exploitation, mais comporte deux touches à enfoncer, dont la touche entrée.

   Sous Spyder : passez avec votre souris sur les icones de la barre des taches et cliquez sur celle qui indique "Run current cell."

   Vous devez alors voir apparaître dans le shell :

```
>>> (executing cell "Cellule 1" (line 2 of "TP1
   eleves 2024-25.py"))
Vous avez bien compilé la cellule 1.
```

   Ici `"Cellule 1"` *est le titre que vous avez mis après les* `##`*, et* `line 2` *est le numéro de la ligne qui débute votre cellule (évidemment ça ne sera pas "2" chez vous.)*
2. Exécutez la deuxième cellule de préférence par raccourci clavier. Vous gagnerez du temps à utiliser cette méthode !

Arguments, imbrications et retours

Exercice 6: [Correction] Nombre d'arguments, arguments optionnels

Quand on écrit un programme sous Python, il est fréquent de devoir / vouloir utiliser des fonctions. Généralement, elles peuvent avoir 0, 1 ou plusieurs arguments nécessaires à son fonctionnement. Par exemple :
- `random` de la bibliothèque `random` n'a aucun argument. Elle s'utilise avec `random()`.
- `int` a un argument.
- `round` n'a par défaut qu'un seul argument et elle rend un arrondi entier de celui-ci. Dans l'exemple donné plus haut, on citait le fait que `round(1.265)` rendait `1`. *(Tester sur plusieurs valeurs si ce n'est pas déjà fait.)*

Toutefois, elle est construite de manière à pouvoir rajouter un argument si on le souhaite : le nombre de chiffres après la virgule que l'on souhaite obtenir pour l'arrondi. Ainsi, `round(1.265,1)` rendait `1.3` Vous pouvez construire vous-même ce type de fonctions. En effet, il suffit dans la définition de la fonction, de rajouter une valeur aux arguments que vous souhaitez optionnels. Par exemple, dans la définition de la fonction `nom_fonction` ci-dessous, l'argument `arg1` est nécessaire et l'argument `arg2` est optionnel et vaudra 10 par défaut :

```
def nom_fonction(arg1, arg2=10):
    ...
```

On pourra donc faire appel à cette fonction soit de la manière suivante :

```
nom_fonction(3):
```

où `arg1` prendra la valeur 3 et `arg2` prendra automatiquement la valeur 10.
ou alors on peut y faire appel de la manière suivante :

```
nom_fonction(3,5):
```

où `arg1` prendra la valeur 3 et `arg2` prendra la valeur 5.

1. Construire une fonction `coupe` qui prend :
- comme argument obligatoire un réel positif `x`
- comme argument optionnel `nb_chiffres`
et qui rend :
- la partie entière de `x` si on fait appel à `coupe(x)`
- le nombre décimal qui correspond à `x` dont on a gardé que les `nb_chiffres` après la virgule si on a fait appel à `coupe(x,nb_chiffres)`
par exemple,

```
import numpy
x=numpy.pi
y=coupe(x)
```

donnera

```
y=3 # ou y=3.0  suivant votre méthode
```

alors que

```
import numpy as np
x=np.pi
y=coupe(x,2)
```

donnera

```
y=3.14
```

(On rappelle à titre d'exemple que $\lfloor pi \times 100 \rfloor = 314$ où $\lfloor x \rfloor$ est la partie entière de x...)

Exercice 7: [Correction]

Les fonctions sont souvent écrites pour être intégrées dans des scripts plus grands ou être utilisées à l'intérieur d'autres fonctions. Cette imbrication est couvent mal maîtrisée par les élèves. On propose donc ici de travailler un peu ce point avec des fonctions simples.
Le but ici est d'effectuer le calcul élémentaire suivant en respectant l'ordre des priorités en mathématiques :

$$x \times (x+y)^2$$

avec x et y deux valeurs donnée en argument à la fonction. Une fonction directe évidente permettant de faire ceci est la suivante :

```
def calcul(x,y):
    resu=x*(x+y)**2
    return(resu)
```

ou même

```
def calcul(x,y):
    return(x*(x+y)**2)
```

mais bien entendu, ce n'est pas ce que je vous demande de faire ici ! Afin de travailler sur les imbrications de fonctions, nous allons ici détailler les opérations. *Évidemment, nous sommes d'accord que les fonctions que je vous demande de construire ci-dessous n'ont aucun intérêt dans la programmation. Elles ne srevent ici que dans un cadre de travail !*

1. Écrire la fonction `somme` qui prend en argument `x` et `y` et qui rend la somme des deux nombres. *(Notez au passage que le fait de définir celle-ci redéfinie celle de l'exercice 2... Vous ne pouvez pas avoir deux fonctions qui portent le même nom !)*

2. Écrire la fonction `produit` qui prend en argument `x` et `y` et qui rend le produit des deux nombres.

3. Écrire maintenant la fonction `calcul_detail(x,y)` qui rend le calcul demandé au départ, mais ceci sans utiliser de symbol mathématique (ni +, ni * ni carré **2) mais seulement en imbriquant correctement les 2 fonctions précédentes. Essayez la et vérifiez le résultat !

Exercice 8: [Correction] Un peu d'opérateurs logiques : une fonction qui rend True ou False

1. Écrire une fonction `divisible5(n)` qui prend en argument un réel quelconque et rend `True` si n est un entier multiple de 5
ou `False` sinon.

2. *Bonus :* Réécrire cette fonction en n'utilisant pas de `if`.

La récursivité : des fonctions qui s'imbriquent dans elles-même

On rappelle qu'une fonction récursive est une fonction qui fait appel à elle même.

Exercice 9: [Correction] Réflexion autour de la fonction factorielle en récursif

La fonction factorielle existe dans la bibliothèque `math` sous le nom `factorial`. On cherche ici à la reproduire. Elle prend un nombre entier n et calcule

$$n! = n \times (n-1) \times \ldots \times 1$$

Récursivement, il faut interpréter ceci de la manière suivante :

$$n! = n \times (n-1)! \quad \forall n \in \mathbb{N}^*$$

Algorithmiquement, cela correspond à *(ne pas tester la fonction ci-dessous)*

```
1  def factoriel(n):
2      return(n*factoriel(n-1))
```

Mais un problème de taille se pose ! Mettons-nous à la place du programme et traitons le "à la main" (souvent utile !) :
On entre par exemple $n = 2$. Le programme fait

$$2 \times \texttt{factoriel(1)}$$

Il garde cette action en mémoire. Il lui faut d'abord calculer `factoriel(1)`, il fait

$$1 \times \texttt{factoriel(0)}$$

Il garde cette action en mémoire. Il lui faut d'abord calculer `factoriel(0)`, il fait

$$0 \times \texttt{factoriel(-1)}$$

Il garde cette action en mémoire. Il lui faut d'abord calculer `factoriel(-1)`, il fait

$$0 \times \texttt{factoriel(-2)}$$

OUPS ! Non seulement il se met à calculer des factoriels négatifs, mais de surcroit, il ne s'arrête jamais ! *(à méditer !)*
Il faut donc lui donner un moment d'arrêt. Pour respecter la définition de "factoriel", ce sera en $n = 0$:

```
1  def factoriel(n):
2      if n==0:
3          return(1)
4      else:
5          return(n*factoriel(n-1))
```

1. Essayer le programme et le commenter. (Pas à l'oral... mais sur votre programme !)
2. Constatez que si n n'est pas un entier naturel positif, ce programme ne s'arrête pas non plus. C'est ennuyeux. *(La commande "Ctrl+I" peut s'avérer utile en cas de fonction qui boucle...)*
   Construire une nouvelle fonction `factoriel2` à partir de la précédente, de manière à ce que, si l'utilisateur ne rentre pas un n correct, le programme signale une erreur, redemande un autre entier n autant de fois qu'il le faut jusqu'à obtenir un n correct et ensuite calcule $n!$.
3. la fonction factorielle peut tout à fait se construire en "itératif" (c'est à dire simplement avec une boucle for ou while). Essayez. Pour les besoins de l'exercice, je vais l'appeler ici `factoriel3`. (On ne se préoccupera pas ici de la validité de n.)
4. On va maintenant comparer la vitesse d'exécution des trois fonctions `factorial`, `factoriel` et `factoriel3`. Pour ceci, nous disposons de la fonction `time` disponible dans le module `time`. On peut par exemple l'importer de la manière suivante :

   ```
   from time import time
   ```

   La fonction `time` (écrite simplement `time()`) rend le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit jusqu'à l'instant auquel elle est utilisée. Ce temps d'exécution n'est pas toujours identique à chaque essai de la fonction. Pour avoir des résutats probants, il faut donc faire un nombre d'essai non négligeable. On propose ici 100 essais.
   Construire une fonction d'entrée n permettant de rendre le temps d'exécution total des trois fonctions sur $N = 100$ essais. Observer le phénomène avec différents n et constatez que finalement ici, la fonction récursive n'est pas forcément la plus intéressante.

La récursivité n'a en réalité que peu d'intérêt pour des problèmes simples de ce type, mais prend toute son importance quand on a affaire à des problèmes plus complexes où l'écriture directe s'avère compliquée. On en verra un exemple dans le TP sur les Tris. On va donc essayer un petit exercice d'entraînement afin de maîtriser un peu mieux la technique :

Exercice 10: [Correction] *(pour s'entraîner un peu)*

Construire une fonction `binome(k,n)` permettant de calculer par récursivité $\binom{n}{k}$ grâce à la formule de récurrence

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \forall n \geqslant 2,\ 1 < k < n$$

On rappelle de plus que $\binom{n}{0} = \binom{n}{n} = 1$ pour tout $n \in \mathbb{N}$ et on suppose qu'on a bien rentré en argument $n \in \mathbb{N}$ et $k \in [\![0, n]\!]$.

FE 1 - Retour sur les basiques

Solutions

Exercice 1

```
n=5
a=3

import math as mt
s=0
i=0
for i in range(1,n+1):
    s+=mt.sqrt(1+a*i)
print('Pour n='+str(n),'le résultat est',str(s) )
```

Exercice 2

```
a=float(input('Donner un réel a>=0 : '))
n=float(input('Donner un nombre entier n : '))

while a<0 or n!=int(n):

    if a<0:
        print("Problème. Il faut a>=0. \
           nRecommencez.")
        a=float(input('Donner un réel a>=0 : '))

    if n!=int(n):
        print("Problème. Il faut n entier positif.\
           nRecommencez.")
        n=float(input('Donner un nombre entier n :
           '))

n=int(n)
s=0
i=0
for i in range(1,n+1):
    s+=mt.sqrt(1+a*i)
print('Pour n='+str(n),'le résultat est',str(s) )
```

Exercice 3

```
import math as mt

def somme(a,n):
    if a/n**2<-1:
        print("Problème. Il faut a>-1. \
           nRecommencez.")
        return(0)
    elif n!=int(n) or n<1:
        print("Problème. Il faut n entier
           strictement positif.\nRecommencez.")
        return(0)
    else:
        s=0
        i=0
        while i<n:
            i+=1
            s+=mt.log(1+a/i**2)
        return(s)
```

Exercice 4

1.

```
import math as mt

def somme(a,n):
    """a un réel strictement plus grand que 1. n un
        entier. Rend la somme pour k allant de 1 à
       n de ln(1+a/n**2)."""
    if a<=-1:
        print("Problème. Il faut a>-1. \
           nRecommencez.")
        return(0)
    elif n!=int(n):
        print("Problème. Il faut n entier.\
           nRecommencez.")
        return(None)
    else:
        s=0
        i=0
        while i<n:
            i+=1
            s+=mt.log(1+a/i**2)
        return(s)
```

2.

```
import math as mt

def somme(a,n):
    """a un réel strictement plus grand que 1. n un
        entier. Rend la somme pour k allant de 1 à
       n de ln(1+a/n**2)."""
    if a<=-1:
        print("Problème. Il faut a>-1. \
           nRecommencez.")
        return(0)
    elif n!=int(n): # on vérifie si n est entier
        print("Problème. Il faut n entier.\
           nRecommencez.")
        return(None)
    else:
        s=0 # Initialisation du s : représente la
           somme des nombres.
        i=0 # Initialisation du i : indice de la
           somme
        while i<n:
            i+=1
            s+=mt.log(1+a/i**2) # on ajoute le ième
                terme.
        return(s)
```

Exercice 6

1.

```
def coupe(x,nb_chiffres=0):
    '''Pour un réel positif x, rend la valeur de x
       avec seulement ses nb_chiffres premières
       décimales.'''
    aux=int(x*10**nb_chiffres) # on va couper les
       chiffres qui ne nous intéressent pas
    return(aux/(10**nb_chiffres))
```

Exercice 7

1.

```
def somme(x,y):
    return(x+y)
```

2.

```
def produit(x,y):
    return(x*y)
```

3.

```
def calcul_detail(x,y):
    S=somme(x,y)
    P=produit(S,S)
    return(produit(x,P))
```

Exercice 8

1.

```
def divisible5(n):
    '''Dit si n est un entier divisible par 5.'''
    if n%5==0:
        return(True)
    else:
        return(False)
```

2.

```
def divisible5Alternative(n):
    '''Dit si n est un entier divisible par 5.'''
    return(n==int(n) and n//5==n/5)
```

Exercice 9

1.

```
def factoriel(n):
    '''Argument : un entier n. Rend n!'''
    if n==0:
        # 0!=1
        return(1)
    else:
        # n!=n*(n-1)!
        return(n*factoriel(n-1))
```

2. Une possibilité peut être donnée par récursivité comme ci-dessous, mais on peut également se débrouiller sans.

```
def factoriel(n):
    '''Argument : un entier n. Calcule n!'''
    if n<0 or type(n)!=int:
        print("Votre argument n'est pas bon. Il
            doit être entier et positif. Recommencez
             :")
        n=flaot(input('n='))
        factoriel(n)
    if n==0:
        # 0!=1
        return(1)
    else:
        # n!=n*(n-1)!
        return(n*factoriel(n-1))
```

3.

```
def factoriel3(n):
    resu=1 # initilisation du résultat final : ici
        correspond à n=0 : 0!=1
    for i in range(1,n+1): # on commence à i=1 car
        0! est déjà donné
        resu=i*resu #i! = ix(i-1)!
    return(resu)
```

4. On peut procéder de la manière suivante :

```
def temps_exe(n):
    """Calcule la différence entre les temps d'
        execution de factoriel et factoriel3."""
    a=time()
    factoriel(n)
    b=time()
    c=b-a
    a=time()
    factoriel3(n)
    b=time()
    c3=b-a
    return(c3-c)
```

On obtient que notre fonction itérative est légèrement plus rapide que notre fonction récursive. On peut peut être expliquer par le fait que la fonction récur-

sive fait systématiquement une vérification à chaque fois que n diminue, alors que `factoriel3` ne fait qu'une seule vérification. Quant à la fonction de Python, elle est évidemment plus rapide que les notres. Elle est probablement construite avec des techniques plus rapide en terme d'espace mémoire.

Exercice 10

```
1  def binome(k,n):
2      if n==0 or k==0 or k==n:
3          return(1)
4      else:
5          return(binome(k-1,n-1)+binome(k,n-1))
```