Algo & Prog, avec Python
(L1-Sciences)
TP n° 2, Automne 2016

Exercice 2.1 Programmez la fonction prenant un entier n ≥ 0 et retournant la factorielle n! de n. Par exemple, 5! = 120

a) Par récurrence, sous la forme d'une fonction `fac_rec(n)`, en supposant le problème résolu pour n-1. Testez sur `fac_rec(5)` qui vaut 120.

b) Par une itération (boucle `while`) sous la forme d'une fonction `fac(n)`. Testez sur `fac(5)`.

c) Python est-il capable[1] de calculer 1000! avec chacune de ces définitions ?

d) Pour chronométrer un calcul en Python, il suffit d'importer la fonction `time()` du module `time`. Le résultat de `time()` est un nombre en secondes flottantes depuis une date arbitraire, seule compte la différence entre deux appels à `time()` pour mesurer une durée écoulée. Exemple :

```
from time import time          # importation de la fonction time()
start = time()                 # top chrono !
f = fac(1000)                  # le calcul proprement dit
end = time()                   # stop chrono !
print('f=',f)                  # affichage du résultat
print('Time =',end-start,'s')  # affichage du temps de calcul (sec)
```

Qui est le plus rapide entre `fac_rec(900)` et `fac(900)` ?

e) Comparez les temps de calcul de n! en prenant n = 10000, puis n = 20000. Quelle vous semble être la loi de formation $t = t(n)$ du temps de calcul en fonction de n ?

f) Quel est le nombre de multiplications nécessaires pour calculer 10000! ? Et pour calculer 20000! ?

g) Les résultats de e) et f) sont-ils contradictoires ?

Exercice 2.2 Reprenons le cours pages 11-12 sur les *nombres premiers*. La fonction `est_premier(n)` programmée en page 12 retourne `False` si elle trouve un *diviseur non trivial* (⇔ dans [2,n-1]) de n. Nous voudrions connaître ce diviseur !

a) Programmez avec une boucle une fonction `ppdiv(n)` prenant un entier n ≥ 2 et retournant son plus petit diviseur ≥ 2. Par exemple `ppdiv(35) == 5`. *Notez au passage que le résultat de `ppdiv(n)` est le plus petit facteur premier de n [pourquoi ?].*

b) En déduire une nouvelle version en une ligne de la fonction `est_premier(n)`.

c) Faites afficher sur une ligne tous les nombres premiers jusqu'à 100.

d) Il est souvent intéressant d'*accélérer un algorithme*. Accélérons donc `ppdiv`. Montrez par un raisonnement mathématique très simple que s'il n'existe aucun diviseur de n dans $[2,\sqrt{n}]$ alors `ppdiv(n) == n`, ce qui évite de chercher dans le gros intervalle $[\sqrt{n},n]$.

e) Implémentez cette optimisation. Au passage, accélérez encore en évitant de considérer les nombres pairs qui à part 2, ne sont pas premiers.

---

[1] On peut faire calculer `fac(5000)` par récurrence en Python en utilisant la fonction `sys.setrecursionlimit(...)` mais il n'est pas toujours clair de trouver la bonne limite...

Exercice 2.3 Définissez en Python la variable `a` dont la valeur est $\sqrt{3}$.

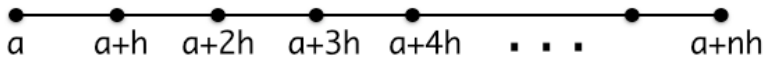
a) Demandez si `a**2` est bien égal à `3`.
b) Expliquez…

Exercices complémentaires (travail personnel, important)

Exercice 2.4 a) Définissez en Python la fonction $x \mapsto f(x) = \dfrac{\sin x}{\sqrt{x^4+1}}$.

b) Calculez une valeur approchée de la dérivée seconde $f''(\sqrt{2})$. Réponse : `0.036705…`

Exercice 2.5 Une définition élémentaire (au lycée) de l'intégrale d'une fonction `f` continue sur un intervalle `[a,b]` consiste à découper `[a,b]` en `n` sous-intervalles de largeur `h` :



puis à ne considérer les valeurs de `f` que sur les points `x=a+ih (0 ≤ i ≤ n-1)`. Lorsque `h` est voisin de `0` (par exemple `h = 0.001`), une approximation de l'intégrale de `f` sur `[a,b]` est donnée par la **formule de Riemann**[2] :

$$\int_a^b f(x)\,dx \approx h\sum_{i=0}^{n-1} f(a+ih)$$

a) Quelle est la signification géométrique de cette formule en terme d'aires ? Dessinez...

b) Programmez en Python la fonction `integrale(f,a,b,n)` retournant cette valeur approchée de l'intégrale de `f` sur `[a,b]` avec `n` sous-intervalles.

c) A.N. Calculez des valeurs approchées de :

$\int_0^1 x^2\,dx$ (Rép: 0.3333...)

$\int_0^2 \dfrac{\sin(x)}{\sqrt{x^4+1}}\,dx$ (Rép: 0.8151...)

m = la valeur moyenne de $\sin(x^2)$ sur $[0,\pi]$ (Rép: 0.2459...)

$\int_0^{\pi} \dfrac{\sin(x)}{\sqrt{x}}\,dx$ (Rép: 1.7896...) *Attention à la singularité en 0 !*

N.B. *Au moment d'utiliser la fonction* `integrale(f,a,b,n)`*, il n'est pas nécessaire que la fonction f soit déjà définie. On peut passer à* `integrale` *une «* ***fonction anonyme*** *». Par exemple, la fonction* $x \mapsto x^5+1$ *qui n'a aucun nom s'écrit en Python :*

```
lambda x : x**5 + 1
```

On pourra donc demander par exemple : `integrale(lambda x : x**5+1, -2, 4, 5000)`

---

[2] L'intégration suivant **Riemann** n'est qu'une première approche, assez naïve. Vous verrez plus tard en maths (L3) la théorie de l'intégrale de **Lebesgue** qui la généralise aux *fonctions mesurables* (pour parler vite, on découpe Oy au lieu de Ox)...

Exercice 2.6 ***Section Math-Info uniquement. Mais rien n'interdit de...***

On considère un système dépendant du temps $t \geq 0$ dont la sortie est donnée par une fonction f vérifiant l'équation différentielle $f'(t) = 1 - f(t)$ avec la condition initiale $f(0) = 0$.

a) En revenant à la définition d'une dérivée et en considérant un très petit accroissement dt du temps, trouvez une formule approchant la valeur *f(t + dt)* connaissant la valeur *f(t)*.

b) A l'aide d'une boucle, itérez 1000 fois la transformation $t \mapsto t + dt$ avec dt = 0.1 et constatez que les valeurs *f(t)* semblent converger vers une limite. Laquelle ?

Algo & Prog, avec Python
(L1-Sciences)
TD n° 2, Automne 2016

Un ordinateur c'est aussi fait pour calculer…

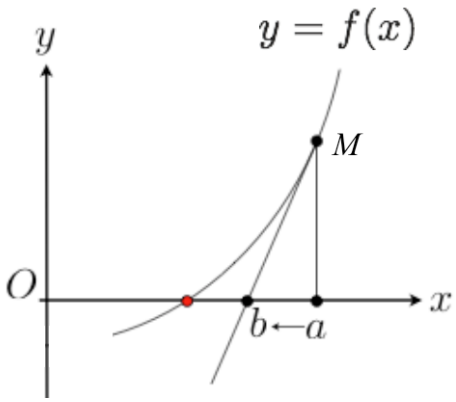
Exercice 2.1 Ecrivez un programme itératif faisant afficher toutes les factorielles des entiers de 0 à 20, une par ligne.

a) En utilisant la fonction `factorial` du module `math`. Combien votre programme fera-t-il de multiplications ?

b) Sans utiliser cette fonction. Tâchez de faire baisser le nombre de multiplications ! Combien votre programme fera-t-il de multiplications ?

Exercice 2.2 Le cours de maths de seconde année vous démontrera peut-être la formule suivante : $\sum_{n=1}^{+\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$. Comment écririez-vous un programme Python permettant d'en déduire une valeur *approchée* de π ?

Exercice 2.3 ***La méthode de Newton pour résoudre des équations f(x) = 0***



Les calculettes modernes possèdent souvent une touche *Solve* permettant de calculer une racine d'une équation $f(x) = 0$. Par exemple, il est difficile sans machine de trouver une solution réelle à l'équation $x^5 - 3x + 1 = 0$.

a) Pourquoi sommes-nous sûrs qu'il y en a au moins une ?

b) Nous allons faire abstraction de la fonction f, la supposer dérivable et à dérivée non nulle *presque partout* (de sorte que la tangente à la courbe existe avec une probabilité quasi-nulle d'être horizontale) pour appliquer la **méthode des tangentes de Newton** vue en cours. On suppose que l'approximation courante est a > 0. Calculez l'équation de la droite tangente à la courbe de f au point $M(a, f(a))$.

c) Calculez la valeur `b` qui est une amélioration de `a` (intersection de la tangente avec Ox). *Si f était la fonction $x \mapsto x^2$ - r du cours, on retrouverait la formule b = 0.5(a + r/a).*

d) Ecrivez la condition pour que l'approximation courante `a` soit suffisamment proche de la solution. On nommera `h` la constante > 0 qui gouverne la précision.

e) Ecrivez une expression mathématique utilisant f, a et h, qui approche la valeur de la dérivée $f'(a)$ de f au point a.

f) Programmez en Python la fonction `solve(f,a,h)` retournant une approximation d'une solution de $f(x)=0$ en partant de l'approximation `a`. L'argument `h` gouvernera la précision.

N.B. *Les exercices 2.3 et 2.4 du TD et le 2.2 du TP sont TRES IMPORTANTS à bien assimiler. Vous devez pouvoir les reproduire sans hésitation un jour de contrôle ou d'examen ! Si la programmation était une musique*[3]*, ces exercices en seraient des gammes…*

Exercice 2.4 *APPLICATIONS NUMERIQUES A FAIRE SUR ORDINATEUR (CHEZ VOUS OU EN TP).*

Utilisez la fonction `solve(f,a,h)` de l'exercice précédent pour faire afficher une valeur approchée de $\sqrt{2}$, puis de $\sqrt[3]{2}$, puis d'une solution de l'équation $x^5-3x+1=0$, puis de l'équation $x=\cos(x)$, et enfin du nombre π. Ouf.

N.B. *Au moment d'utiliser la fonction* `solve(f,a,h)`*, il n'est pas nécessaire que la fonction f soit déjà définie. On peut passer à* `solve` *une « **fonction anonyme** ». Par exemple, la fonction* $x \mapsto x^2+1$ *- sans qu'il soit besoin de la définir - s'écrit en Python :*

```
lambda x : x**2 - 1
```

On pourra donc demander par exemple : `solve(lambda x : x**2 - 1, 3, 0.01)`

Exercices complémentaires sur la stratégie de Newton

Exercice 2.5 En cours, nous avons appliqué la stratégie des tangentes de Newton au calcul approché de la racine carrée. Sans utiliser la fonction `solve` de l'exercice 2.3, trouvez `approx`, `assez_bonne` et `améliore` pour le problème du calcul de la **racine cubique** de `r`.

```
>>> rac3(2,0.001)
1.259933493449977
```

```
>>> 2**(1/3)
1.2599210498948732
```

Exercice 2.6 La stratégie de Newton consiste à appliquer la fonction `améliore` - notons-la `f` pour raccourcir - suffisamment de fois pour atteindre la précision souhaitée en calculant les itérées `a`, `f(a)`, `f(f(a))`, `f(f(f(a)))`, `f(f(f(f(a))))`, etc. Les matheux savent démontrer que cette suite converge - sous de bonnes conditions - vers un **point fixe** s de la fonction `f`, qui vérifie `f(s) = s`. Un autre point de vue consisterait alors à définir une **fonction générale de calcul de point fixe** et d'en déduire une nouvelle stratégie :

```
def point_fixe(f,a,h) :
    ...
```

```
def rac3(r,a,h) :  # ou solve, etc
    def améliore(x) :
        ...
    return point_fixe(améliore,a,h)
```

Nous vous laissons définir la fonction `point_fixe`. Indication : calculez les itérées `a`, `f(a)`, `f(f(a))`, etc jusqu'à ce que la différence entre deux itérées consécutives soit inférieure à la précision `h`. *Notez au passage que nous n'utilisons plus la dérivée pour résoudre cos(x) = x...*

[3] *La programmation est une musique* ☺.