Informatique — MPSI/PCSI

Correction du TP n° 9

Exercice 9.1
Avec ce qui est donné dans le sujet, il vient :

```
def f(x):
    """

    Argument :

        x (float) : réel positif

    Résultat :

        (float) : réel positif, racine de x

    """
    assert isinstance(x, (int,float)), "x doit être un nombre"
    r = x**2
    assert (isinstance(r, (int,float)) and r>=0), \
    "le résultat doit être un nombre positif"
    return(r)
```

Exercice 9.2
Dans le cas où il y a une exception, la clause `except AssertionError` permet de renvoyer le message d'erreur et de poursuivre la boucle. La première exception est levée avec la chaîne de caractère `"a"`. Comme dans ce cas `f(x)` n'est pas évaluée (à cause du problème d'assertion), la valeur de `y` reste inchangée et ainsi, lorsque `x` vaut `"a"`, il est affiché la valeur correspondant à `f(2.3)`. En mettant une directive `continue`, le code saute l'affichage potentiellement erroné de la valeur et passe directement à la valeur de `x` suivante.

Exercice 9.3 (Implémentation du logarithme népérien)

1. Par définition, $\ln : \mathbb{R}_*^+ \to \mathbb{R}$ donc on doit avoir $x > 0$. Comme les nombres réels sont représentés par des flottants, x doit être un flottant strictement positif. Comme $\mathbb{N}_* \subset \mathbb{R}_*^+$ alors, il peut aussi être un naturel. La spécification de la fonction peut alors être décrite par l'aide suivante :

```
def ln(x, epsilon=1e-8):
    """ Calcule une approximation à espilon près du réel correspondant
    au logarithme népérien de x.

    Argument :

        x (float) : réel positif ou naturel

    Argument optionnel :

        epsilon (float) : niveau de précision souhaité, par défaut à 1e-8

    Résultat :

        (float) : logarithme népérien de x

    """
```

   suivie de la pré-condition :

```
    assert isinstance(x, (int,float)) and (x>0), \
        "x doit être un réel ou un naturel strictement positif"
    assert isinstance(epsilon, float) and (epsilon>0) and (epsilon<1)
        "epsilon doit être un réel positif strictement plus petit que 1"
```

2. Au voisinage de $x = 0$, on a :

$$\ln(1+x) = \sum_{k=1}^{n} (-1)^{k+1} \frac{x^k}{k} + o\left(x^n\right)$$

Cette somme ne converge que pour $|x| \leqslant 1$; d'où :

$$\forall x \in [1;2], \quad \ln(x) = \sum_{k=1}^{n} (-1)^{k+1} \frac{(x-1)^k}{k} + o\left((x-1)^n\right)$$

En prenant n tel que

$$\left| \frac{(x-1)^{n+1}}{n+1} \right| < \varepsilon$$

alors

$$\sum_{k=1}^{n} (-1)^{k+1} \frac{(x-1)^k}{k}$$

est une approximation à ε près de $\ln(x)$.

3. Dans le cas $x \geqslant 2$, il suffit de s'appuyer sur l'égalité

$$\ln(x) = 2^n \ln\left(\sqrt[n]{x}\right)$$

que l'on peut écrire de façon récursive

$$\ln(x) = \begin{cases} \ln(x) & \text{si } 1 \leqslant x < 2 \\ 2\ln(\sqrt{x}) & \text{sinon} \end{cases}$$

4. En utilisant l'égalité $\ln(x) = -\ln(1/x)$ dans le cas $x \in]0,1[$, il vient :

$$\ln(x) = \begin{cases} \displaystyle\sum_{k=1}^{n} (-1)^{k+1} \frac{(x-1)^k}{k} \quad \text{avec} \quad \frac{(x-1)^{n+1}}{n+1} < \varepsilon & \text{si } 1 \leqslant x < 2 \\ -\ln(1/x) & \text{si } 0 < x < 1 \\ 2\ln(\sqrt{x}) & \text{sinon} \end{cases}$$

En veillant à minimiser le nombre de calculs pour évaluer la somme

$$\sum_{k=1}^{n} (-1)^{k+1} \frac{(x-1)^k}{k}$$

dans le cas $x \in [1;2]$, sachant $x^k = x^{k-1} \times x$, et à diviser par deux le niveau de précision de chaque appel récursif dans le cas $x \geqslant 2$, il vient :

```
def ln(x, epsilon=1e-8):
    """ Calcule une approximation à espilon près du réel correspondant
    au logarithme népérien de x.

    Argument :

        x (float) : réel positif ou naturel

    Argument optionnel :

        epsilon (float) : niveau de précision souhaité, par défaut à 1e-8
```

```
    Résultat :

        (float) : logarithme népérien de x

    """
    assert isinstance(x, (int,float)) and (x>0), \
        "x doit être un réel ou un naturel strictement positif"
    assert isinstance(epsilon, float) and (epsilon>0) and (epsilon<1), \
        "epsilon doit être un réel positif strictement plus petit que 1"
    if x<1:
        retrun(-ln(1/x))
    elif x>=2:
        return(2*ln(x**.5, epsilon/2))
    else:
        k=1     # initialisation de la puissance
        x=x-1   # changement de variable pour calculer ln(1+x)
        a=x     # initialisation ajout
        r=x     # initialisation résultat
        t=1     # initialisation du (-1)^{k+1}
        xk=x    # initialisation x**k
        while abs(a)>=epsilon:
            k+=1
            t=-t    # (-1)^{k+1} = (-1)*(-1)^k
            xk*=x   # x^k = x^{k-1} * x
            a=t*xk/k # ajout
            r+=a
        return(r)
```

5. On commence par s'appuyer sur la comparaison avec un résultat supposé correct renvoyé par la fonction `log` de la bibliothèque `math` en définissant une fonction :

```
def erreur(x,epsilon):
    import math as ma
    return(abs(ma.log(x)-ln(x,epsilon))<epsilon)
```

qui renvoie le booléen `True` si l'approximation est correcte, `False` sinon. Comme notre implémentation de la fonction ln nécessite trois domaines distincts, on veillera à tester des valeurs dans ces trois domaines. De même, comme l'implémentation dépend du niveau de précision souhaité, on veillera aussi à faire varier les niveaux de précision. Le code suivant permet de vérifier

```
def tests():
    dirinit = dir()+['dirinit']
    def erreur(x,epsilon):
        import math as ma
        return(abs(ma.log(x)-ln(x,epsilon))<epsilon)
    for epsilon in [1e-6, 1e-8, 1e-12]:
        for x in [0,1,1.2,3,12,120,5000,5000000]:
            try:
                # on s'assure que l'on puisse calculer
                try:
                    ln(x, epsilon)
                except AssertionError as msg:
                    print("*** x="+str(x)+", epsilon="+str(epsilon)+":", msg)
                    continue # pour passer si l'exception vient de ln
                assert erreur(x,epsilon), "x="+str(x)+", epsilon="+str(epsilon)
                print("pass x="+str(x)+", epsilon="+str(epsilon))
            except:
                print("DO NOT pass x="+str(x)+", epsilon="+str(epsilon))
            finally:
                for e in dir():
```

```
                    if e not in dirinit:
                        del(e)
tests()
```

Exercice 9.4 (Évaluation de polynômes – schéma de Horner)

1. On commence par l'aide suivante qui précise notamment l'ordre des coefficients dans la liste C.

```
def Horner(x,C):
    """ Calcule une approximation du polynôme P de coefficients réels
    C = [c0, c1, c2, ..., cn] en un réel x.

    Arguments :

        x (float) : réel où évaluer P(x)
        C (itérable, list ou tuple) : coefficients réels
            du polynôme P donnés dans l'ordre croissant

    Résultat :

        (float) : approximation de P(x)

    """
```

Pour vérifier que x soit un nombre réel, on écrira simplement :

```
assert isinstance(x, (int,float)), "x doit être un nombre réel"
```

Pour s'assurer que `C` soit une séquence de type `list` ou `tuple`, on effectuera le test

```
isinstance(C, (list,tuple))
```

Pour s'assurer que chacun des éléments de cette séquence soit un nombre, on effetuera le test :

```
not(False in [isinstance(e, (int,float)) for e in C])
```

ce que l'on regroupe sous la forme :

```
assert isinstance(C, (list,tuple)) and \
    not(False in [isinstance(e, (int,float)) for e in C]), \
    "C doit être une séquence (liste ou tuple) de nombres réels"
```

2. Partant de $k = 0$ avec $[c_0, \ldots, c_n]$, et prenant comme variant $n - k$ la longueur de la liste des coefficients restants à utiliser $[c_k, \ldots, c_n]$ et considérant l'invariant il vient :

$$\text{Horner}([c_k, \ldots, c_n], x) = \begin{cases} c_n & \text{si } k = n \\ c_k + x \times \text{Horner}([c_{k+1}, \ldots, c_n], x) & \text{sinon.} \end{cases}$$

ce que l'on peut traduire par :

```
def Horner(x,C):
    """ Calcule une approximation du polynôme P de coefficients réels
    C = [c0, c1, c2, ..., cn] en un réel x.

    Arguments :

        C (itérable, list ou tuple) : coefficients réels
            du polynôme P donnés dans l'ordre croissant
        x (float) : réel où évaluer P(x)

    Résultat :

        (float) : approximation de P(x)

    """
```

```
    assert isinstance(C, (list,tuple)) and \
        not(False in [isinstance(e, (int,float)) for e in C]), \
        "C doit être une séquence (liste ou tuple) de nombres réels"
    assert isinstance(x, (int,float)), "x doit être un nombre réel"
    if len(C)==1:
        return(C[0])
    else:
        return(C[0]+x*Horner(x,C[1:]))
```

Notez que nous aurions aussi pu utiliser la version impérative :

```
def Horner(x,C):
    n=len(C)
    u=C[n-1]
    for k in range(n-1):
        u=x*u+C[n-k]
    return(u)
```

3. On commence par s'appuyer sur la comparaison avec un résultat supposé correct renvoyé par la fonction `polyval` de la bibliothèque `numpy.polynomial.polynomial` en définissant une fonction :

```
def peval(C,x):
        import numpy.polynomial.polynomial as npp
        return(npp.polyval(x, C))
```

qui renvoie un réel. On test ensuite si notre approximation est correcte à ε près pour fabriquer un booléen. Le code suivant permet de vérifier le comportement :

```
def tests():
    dirinit = dir()+['dirinit']
    epsilon = 1e-8
    def peval(C,x):
        import numpy.polynomial.polynomial as npp
        return(npp.polyval(x, C))
    for C in [[1,2,3,4], [1,0,3,-4.2]]:
        for x in [0, 1.2, 3.4]:
            try:
                # on s'assure que l'on puisse calculer
                try:
                    y = Horner(x, C)
                except AssertionError as msg:
                    print("*** x="+str(x)+", C ="+str(C)+":", msg)
                    continue # pour passer si l'exception vient de Horner
                assert abs(peval(C,x)-y)<=epsilon, "x="+str(x)+", C="+str(C)
                print("pass x="+str(x)+", C ="+str(C))
            except:
                print("DO NOT pass x="+str(x)+", C ="+str(C))
            finally:
                for e in dir():
                    if e not in dirinit:
                        del(e)
```

Exercice 9.5 (Exponentiation rapide, arbre des puissances de Knuth)

1. L'invariant est ici la propriété :

$$\forall n \in \mathbb{N}_*, \quad x^n = \begin{cases} x^{2k} = x^k \times x^k & \text{si } n = 2k \text{ (pair)} \\ x^{2k+1} = x \times x^k \times x^k & \text{sinon.} \end{cases}$$

Partant de $k = n$, on écrit simplement :

$$x^k = \begin{cases} x & \text{si } k = 1 \\ \begin{cases} x^{\left\lfloor \frac{k}{2} \right\rfloor} \times x^{\left\lfloor \frac{k}{2} \right\rfloor} & \text{si } k = 2\left\lfloor \frac{k}{2} \right\rfloor \text{ (pair)} \\ x^{\left\lfloor \frac{k}{2} \right\rfloor} \times x^{\left\lfloor \frac{k}{2} \right\rfloor} \times x & \text{sinon.} \end{cases} & \text{sinon.} \end{cases}$$

où l'on voit que le k suivant sera $\left\lfloor \frac{k}{2} \right\rfloor < k$ donc la suite des « k » est bien strictement décroissante, jusqu'à 1. On prend donc comme variant la puissance k, de x^k, variant de n à 1.

2. **(Correction partielle)** Supposant que chaque appel récurssif fournisse un résultat correct, dans ce cas le résultat renvoyé par la fonction `puissance` sera correct car, si n pair,

$$x^{\left\lfloor \frac{n}{2} \right\rfloor} \times x^{\left\lfloor \frac{n}{2} \right\rfloor} = x^n$$

et si n impair

$$x^{\left\lfloor \frac{n}{2} \right\rfloor} \times x^{\left\lfloor \frac{n}{2} \right\rfloor} \times x = x^n$$

car $n = 2\left\lfloor \frac{n}{2} \right\rfloor + 1$. Avec cet invariant, l'algorithme est partiellement correct.
(Terminaison) Ayant pris comme variant l'entier $k = \left\lfloor \frac{n}{2} \right\rfloor$, strictement décroissant de n à 1, l'algorithme se termine toujours avec un appel où $n = 1$. Dans ce cas la valeur retournée est $x^1 = x$.
(Correction) Comme l'algorithme est partiellement correct (invariant) et qu'il termine (variant), alors il est correct.
(Complexité) Pour évaluer la complexité, on se limite à l'évaluation du nombre de multiplications : 1 dans le cas pair, 2 dans le cas impair. En majorant par 2, sachant que le nombre d'appels récurssifs est forcément inférieur ou égal à $\log_2(n)$, on en déduit une majoration :

$$2\log_2(n) = O(\ln(n))$$

3. En prenant soin de considérer des nombres réels pour x et des entiers positifs pour n, il vient le code suivant.

```
def puissance(x,n):
    """Calcule x^n, le nombre x àla puissance n,
    avec le moins de calculs possible.

    Arguments :

        x (float) : nombre réel
        n (int) : entier naturel, puissance

    Résultat :

        (float) : approximation de x**n
    """
    assert isinstance(x, (int,float)), "x doit être un nombre réel"
    assert isinstance(n, int) and (n>=0), "n doit être un entier naturel"
    if n==1:
        return(x)
    else:
        z=puissance(x,n//2)
        if n%2==0:
            return(z*z)
        else:
            return(x*z*z)
```

4. On commence par s'appuyer sur la comparaison avec un résultat supposé correct renvoyé par la fonction `pow`. Le code suivant permet de vérifier le comportement :

```
def tests():
    dirinit = dir()+['dirinit']
    def erreur(t):
        return(puissance(t[0],t[1])==pow(t[0],t[1]))
    for t in ((5,13), (2,3), (3,4), (15,11)):
        try:
            # on s'assure que l'on puisse calculer
            try:
                puissance(t[0],t[1])
            except AssertionError as msg:
                print("*** (x,n)="+str(t)+":", msg)
                continue # pour passer si l'exception vient de puissance
            assert erreur(t), "(x,n)="+str(t)
            print("pass (x,n)="+str(t))
        except:
            print("DO NOT pass (x,n)="+str(t))
        finally:
            for e in dir():
                if e not in dirinit:
                    del(e)
```

5. En distinguant les cas pairs $n > 2$ des cas impairs pour lesquels on utilise une boucle `while` pour égrenner les diviseurs impairs potentiels jusqu'à $\lfloor \sqrt{n} \rfloor$, il vient le code suivant :

```
def factorisation(n):
    if n>2:
        if n%2==0:
            s=[2,n//2]
        else:
            k=3
            while k**2<=n and (n%k)!=0:
                k+=2
            if (k**2>n):
                s=[n]
            else:
                s=[k,n//k]
    elif n==1:
        s=[1]
    else:
        s=[2]
    return(s)
```

6. Avec la relation donnée, il vient immédiatement :

```
def puissance_facteur(x,n):
    if n==0:
        return(1)
    elif n==1:
        return(x)
    else:
        s = factorisation(n)
        if len(s)==1:# n est premier
            return(x*puissance_facteur(x,n-1))
        else:
            p,q = s
            return(puissance_facteur(puissance_facteur(x,p),q))
```

Pour tester ce code, il suffit d'adapter le code `tests` précédent en remplaçant la fonction `puissance` par `puissance_facteur`.

7. Pour déterminer le chemin d'une feuille, il suffit d'ajouter chaque feuille vue lors de la remontée pour être à la racine (parent `None`). Le code suivant convient.

```
def chemin(N,i):
    p,C = N[i],[i]
    while p is not None:
        C.append(p)
        p=N[p]
    C.reverse()
    return(C)
```

8. Pour déterminer le niveau d'une feuille, il suffit de compter le nombre de fois qu'il est nécessaire de remonter pour être à la racine (parent `None`). Le code suivant convient.

```
def niveau(N,i):
    p,n = N[i],1
    while p is not None:
        n+=1
        p=N[p]
    return(n)
```

9. Pour construire l'arbre de Knuth, on part du niveau 3 avec le dictionnaire `N` donné dans le sujet. Ayant défini les feuilles du niveau k dans la liste `W`, il suffit de toutes les parcourir avec l'algorithme de Knuth et d'ajouter tous les $n + a_i$ qui n'y sont pas (si `n+a not in N.keys()` est vrai), avec comme parent la feuille n, ce que l'on écrit :

```
for n in W:
    for a in chemin(N,n):
        if n+a not in N.keys():
            N[n+a]=n
```

En tenant compte de l'argument optionnel avec lequel il est nécessaire de déterminer le nombre de niveaux de l'arbre et les feuilles du dernier niveau (méthode du candidat avec ajout si multiples) et de la mise à jour de la liste `W` avec une liste `Z` remplie lors du parcours et contenant les feuilles du niveau $k + 1$, il vient le code suivant :

```
def knuth(z,N=None):
    if N is None:
        N = {1:None,2:1,3:2,4:2}
        k=3
        W = [3,4]
    else:
        Z = [(e,niveau(N,e)) for e in N.keys()]
        k = Z[0][1]
        W = [Z[0][0]]
        for e in Z[1:]:
            if e[1]>k:
                k = e[1]
                W = [e[0]]
            elif e[1]==k:
                W.append(e[0])
    while k<z:
        Z=[]
        for n in W:
            for a in chemin(N,n):
                if n+a not in N.keys():
                    N[n+a]=n
                    Z.append(n+a)
        k+=1
        W=Z.copy()
    return(N)
```

10. La seule difficulté pour déterminer les puissances de Knuth est d'avoir la valeur de n souhaité dans l'arbre. Le code suivant suffit.

```
def knuth_exp(n):
    k=3
    N = knuth(k)
    while N.get(n) is None:
        k+=1
        N = knuth(k,N)
    return(chemin(N,n))
```

11. Pour déterminer une combinaison (a, b) telle que $a + b = s$ en prenant a et b dans `L`, il suffit de parcourir toutes les valeurs et de renvoyer la première combinaison satisfaisante (qui existe si l'arbre de Knuth est correct). C'est ce que fait le code suivant, en partant de la fin avec `[ :-1]` car les combinaisons sont plus probables avec les grandes valeurs de la coupe de liste. C'est ce que fait le code suivant.

```
def combinaisons(L,s):
    for a in L[::-1]:
        for b in L[::-1]:
            if (a+b)==s:
                return(a,b)
```

12. On utilise la mémoïsation dans un dictionnaire `P` pour stocker les valeurs de x^k calculées, qui sont les seules que peut exploiter une combinaison $x^k = x^a \times x^b$ (où $k =$ `E[i]`). Le code suivant convient.

```
def puissance_knuth(x,n):
    E = knuth_exp(n)
    P={1:x}
    for i in range(1,len(E)):
        a,b=combinaisons(E[:i],E[i])
        P[E[i]]=P[a]*P[b]
    return(P[n])
```

* *
*