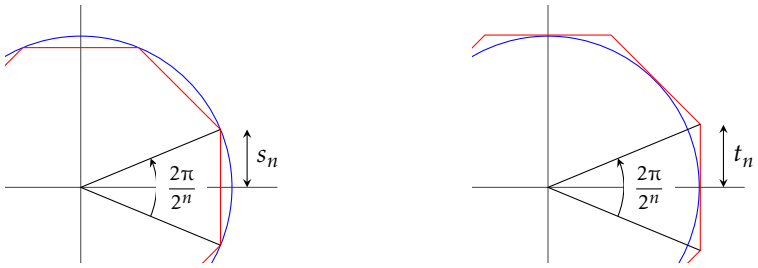
Corrigé

Calculs en précision arbitraire

Calcul de π par la méthode d'Archimède

Question 1. Considérons les demi-longueurs des côtés s_n et t_n représentés sur les dessins ci-dessous :



Nous avons $s_n = \sin\left(\frac{\pi}{2^n}\right)$ et $t_n = \tan\left(\frac{\pi}{2^n}\right)$ donc $u_n = 2^n s_n = 2^n \sin\left(\frac{\pi}{2^n}\right)$, $v_n = 2^n t_n = 2^n \tan\left(\frac{\pi}{2^n}\right)$.

La relation $\sin(2a) = 2\sin(a)\cos(a) = \dfrac{2\sin(a)^2}{\tan(a)}$ conduit à l'égalité $u_n = \dfrac{u_{n+1}^2}{v_{n+1}}$ donc $u_{n+1} = \sqrt{u_n v_{n+1}}$.

La relation $\dfrac{1}{\tan(2a)} + \dfrac{1}{\sin(2a)} = \dfrac{\cos(2a)+1}{\sin(2a)} = \dfrac{2\cos(a)^2}{2\sin(a)\cos(a)} = \dfrac{1}{\tan(a)}$ conduit à l'égalité $\dfrac{1}{v_n} + \dfrac{1}{u_n} = \dfrac{2}{v_{n+1}}$ donc $v_{n+1} = \dfrac{2u_n v_n}{u_n + v_n}$.

Ceci permet d'en déduire la fonction suivante :

```
def approx_pi(n):
    u, v = 2*sqrt(2), 4
    for k in range(n):
        print('pour n = {}, valeur par défaut : {} et par excès : {}'.format(k+2, u, v))
        v = (2*u*v)/(u+v)
        u = sqrt(u*v)
```

Voici ce qu'on obtient pour $n = 40$:

```
pour n =  2, valeur par défaut : 2.8284271247461903 et par excès : 4
pour n =  3, valeur par défaut : 3.0614674589207187 et par excès : 3.3137084989847607
pour n =  4, valeur par défaut : 3.121445152258053  et par excès : 3.1825978780745285
pour n =  5, valeur par défaut : 3.1365484905459398 et par excès : 3.151724907429257
pour n =  6, valeur par défaut : 3.1403311569547534 et par excès : 3.144118385245905
pour n =  7, valeur par défaut : 3.1412772509327733 et par excès : 3.1422236299424577
.....................................................................................
pour n = 23, valeur par défaut : 3.141592653589719  et par excès : 3.141592653589939
pour n = 24, valeur par défaut : 3.141592653589774  et par excès : 3.141592653589829
pour n = 25, valeur par défaut : 3.141592653589788  et par excès : 3.1415926535898016
pour n = 26, valeur par défaut : 3.141592653589791  et par excès : 3.141592653589794
pour n = 27, valeur par défaut : 3.1415926535897922 et par excès : 3.141592653589793
pour n = 28, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
pour n = 29, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
pour n = 30, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
pour n = 31, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
.....................................................................................
pour n = 39, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
pour n = 40, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
pour n = 41, valeur par défaut : 3.1415926535897927 et par excès : 3.1415926535897927
```

On constate que passé $n = 28$ la précision des nombres flottants devient insuffisante : les suites de nombres flottants qui représentent u_n et v_n sont stationnaires.

Question 2. Montrons par récurrence sur $n \geqslant 2$ que $a_n \leqslant 10^{200}u_n \leqslant c_n$ et $b_n \leqslant 10^{200}v_n \leqslant d_n$.

- C'est évident pour $n = 2$.
- Si $n \geqslant 2$, supposons le résultat acquis au rang n et montrons le au rang $n+1$.

  Nous avons $\dfrac{1}{c_n} \leqslant \dfrac{1}{10^{200}u_n} \leqslant \dfrac{1}{a_n}$ et $\dfrac{1}{d_n} \leqslant \dfrac{1}{10^{200}v_n} \leqslant \dfrac{1}{b_n}$ donc $\dfrac{1}{c_n}+\dfrac{1}{d_n} \leqslant \dfrac{1}{10^{200}u_n}+\dfrac{1}{10^{200}v_n} \leqslant \dfrac{1}{a_n}+\dfrac{1}{b_n}$.

  On en déduit que $\dfrac{1}{c_n}+\dfrac{1}{d_n} \leqslant \dfrac{2}{10^{200}v_{n+1}} \leqslant \dfrac{1}{a_n}+\dfrac{1}{b_n}$ donc $\dfrac{2a_nb_n}{a_n+b_n} \leqslant 10^{200}v_{n+1} \leqslant \dfrac{2c_nd_n}{c_n+d_n}$, ce qui implique :

$$b_{n+1} \leqslant 10^{200}v_{n+1} \leqslant d_{n+1}.$$

  De même, $a_n \leqslant 10^{200}u_n \leqslant c_n$ et $b_{n+1} \leqslant 10^{200}v_{n+1} \leqslant d_{n+1}$ donc $a_nb_{n+1} \leqslant 10^{400}u_nv_{n+1} \leqslant c_nd_{n+1}$ ce qui implique :

$$a_{n+1} \leqslant 10^{200}u_{n+1} \leqslant c_{n+1}.$$

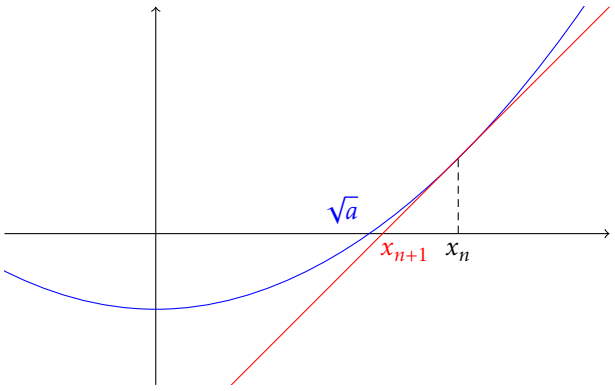
Question 3. La définition de ces deux fonctions est élémentaire :

```
def floor(x, y):
    return x // y

def ceil(x, y):
    if x % y == 0:
        return x // y
    else:
        return x // y + 1
```

Calcul de la racine carrée de grands entiers

La méthode de Newton-Raphson, dite aussi méthode des tangentes, consiste à remplacer l'équation (non linéaire) $f(x) = 0$ par l'équation linéaire approchée $f(c)+(x-c)f'(c) = 0$. On est ainsi conduit à étudier la suite $(x_n)_{n\in\mathbb{N}}$ définie par la relation de récurrence $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$. Dans le cas de la fonction $f : x \mapsto x^2 - a$ on obtient $x_{n+1} = g(x_n)$ avec $g(x) = \dfrac{1}{2}\left(x + \dfrac{a}{x}\right)$.



La convexité de la fonction f (le fait que son graphe soit situé au dessus de chacune de ses tangentes) prouve que si $x_0 > \sqrt{a}$, la suite $(x_n)_{n\in\mathbb{N}}$ décroit et converge vers $\sqrt{a}$.

Lorsque a est entier, on définit une suite d'entiers naturels en posant $r_0 = a$ et $\forall n \in \mathbb{N}$, $r_{n+1} = \lfloor g(r_n) \rfloor$.
Nous avons $r_{n+1} - r_n = \left\lfloor \dfrac{1}{2}\left(\dfrac{a}{r_n} - r_n\right)\right\rfloor$ donc tant qu'on aura $r_n > \sqrt{a}$ nous aurons $\dfrac{1}{2}\left(\dfrac{a}{r_n} - r_n\right) < 0$ et ainsi $r_{n+1} - r_n \leqslant -1$.
Ceci assure l'existence d'un rang N vérifiant $r_N \leqslant \sqrt{a} < r_{N-1}$.
L'inégalité $r_N \leqslant \sqrt{a}$ prouve que $r_N \leqslant \lfloor \sqrt{a} \rfloor$; l'inégalité $r_{N-1} > \sqrt{a}$ prouve que $g(r_{N-1}) > \sqrt{a}$ et donc que $r_N = \lfloor g(r_{N-1}) \rfloor \geqslant \lfloor \sqrt{a} \rfloor$.
Ainsi, nous avons $r_N = \lfloor \sqrt{a} \rfloor$, et cette quantité peut être calculée à l'aide de la fonction :

```
def isqrt(a):
    r = a
    while r**2 > a:
        r = (r**2 + a) // (2 * r)
    return r
```

Remarque. Il est possible d'améliorer les performances de cette fonction en essayant de partir d'une valeur initiale plus proche de $\sqrt{a}$. En notant k le nombre de chiffres de l'écriture décimale de a nous avons $a < 10^k$ donc $\sqrt{a} < 10^{\lceil k/2 \rceil}$, ce qui permet de choisir $r_0 = 10^{\lceil k/2 \rceil}$.
Par ailleurs, on peut observer que le carré de r est calculé deux fois. Or le temps de calcul croît avec le nombre de chiffres de r ; puisqu'on prévoit d'appliquer cette fonction à de très grandes valeurs on a intérêt à n'effectuer ce calcul qu'une fois, ce qui conduit à la version suivante :

```
def fsqrt(a):
    k = len(str(a))
    r = 10**((k + 1) // 2)
    r2 = r * r
    while r2 > a:
        r = (r2 + a) // (2 * r)
        r2 = r * r
    return r
```

Pour obtenir $\lceil \sqrt{a} \rceil$ il suffit d'observer que $\lceil x \rceil = \begin{cases} \lfloor x \rfloor + 1 & \text{lorsque } x \notin \mathbb{Z} \\ \lfloor x \rfloor & \text{sinon} \end{cases}$, soit :

```
def csqrt(a):
    r = fsqrt(a)
    if r * r == a:
        return r
    else:
        return r + 1
```

Calcul des décimales de π

Question 4. Les quatre fonctions que nous venons d'écrire nous permettent maintenant d'itérer les valeurs *exactes* des quatre suite a, b, c et d et ainsi obtenir un encadrement par deux entiers a_n et d_n de $10^{200}\pi$. On choisit de stopper l'itération lorsque $(a_n, b_n, c_n, d_n) = (a_{n+1}, b_{n+1}, c_{n+1}, d_{n+1})$:

```
n = 2
a, c = fsqrt(8*10**400), csqrt(8*10**400)
b = d = 4*10**200

while True:
    next_b = floor(2 * a * b, a + b)
    next_a = fsqrt(a * next_b)
    next_d = ceil(2 * c * d, c + d)
    next_c = csqrt(c * next_d)
    if (a, b, c, d) == (next_a, next_b, next_c, next_d):
        break
    a, b, c, d = next_a, next_b, next_c, next_d
    n += 1

print('Valeur optimale de n :', n)
print('Valeur par défaut de 10**200.pi :', a)
print('Valeur par excès de 10**200.pi :', d)
```

Ce script fournit les résultats suivants :

```
Valeur optimale de n : 335
Valeur par défaut de 10**200.pi : 31415926535897932384626433832795028841971693993751058209749
44592307816406286208998628034825342117067982148086513282306647093844609550582231725359408128
48111745028410270193852110555964462294895493037916
Valeur par excès de 10**200.pi :  31415926535897932384626433832795028841971693993751058209749
44592307816406286208998628034825342117067982148086513282306647093844609550582231725359408128
48111745028410270193852110555964462294895493038369
```

On observe que seuls les quatre derniers chiffres de a_n et d_n diffèrent. Ceci nous donne une approximation de π avec 197 chiffres exacts (soit 196 après la virgule).

Et pour ceux qui s'ennuient

Question 5. Commençons par illustrer la méthode en calculant $\left\lfloor\sqrt{64015}\right\rfloor$:

$$\begin{array}{rr|l}
 & 6.4\,0.1\,5 & 2\,5\,3 \\
\cline{3-3}
-\ & 4 & \textcircled{2}\times 2 = 4 \\
\cline{2-2}
 & 2.4\,0 & 4\,\textcircled{5}\times 5 = 2\,2\,5 \\
-\ & 2\,2\,5 & 5\,0\,\textcircled{3}\times 3 = 1\,5\,0\,9 \\
\cline{2-2}
 & 1\,5.1\,5 & \\
-\ & 1\,5\,0\,9 & \\
\cline{2-2}
 & 6 &
\end{array}$$

Une fois toutes les tranches traitées, on a obtenu $\left\lfloor\sqrt{64015}\right\rfloor = 253$.

Débutons la rédaction de l'algorithme correspondant par une fonction qui découpe en tranches un entier en retournant la liste des tranches :

```
def tranches(n):
    x = n
    lst = []
    while(x>0):
        lst = [x % 100] + lst
        x //= 100
    return lst
```

Poursuivons avec une fonction qui recherche le plus grand entier i vérifiant $(20r+i)\times i \leqslant x$:

```
def cherche(r, x):
    i = 9
    while (20*r+i)*i > x:
        i -= 1
    return i
```

Écrivons enfin la fonction principale :

```
def sqrt(n):
    lst = tranches(n)
    x = r = 0
    while len(lst) > 0:
        x = 100*x+lst.pop(0)
        i = cherche(r, x)
        x -= (20*r+i)*i
        r = r*10+i
    return r
```

Cette dernière fonction réalise l'itération de deux suites r et x. Dans le cas où $n = 64015$, elles prennent successivement les valeurs : $(r_1, x_1) = (2, 2)$, $(r_2, x_2) = (25, 15)$, $(r_3, x_3) = (253, 6)$.
La terminaison de l'algorithme est assurée par le fait que la méthode `pop(0)` supprime le premier élément de la liste à qui elle est appliquée.
Notons $t_1, t_2, \ldots, t_p$ les différentes tranches de n, et prouvons les invariants : $r_k = \left\lfloor\sqrt{[t_1 t_2 \cdots t_k]}\right\rfloor$ et $r_k^2 + x_k = [t_1 t_2 \cdots t_k]$, les crochets désignant ici la concaténation des tranches.
C'est clair pour $k = 0$ car $r_0 = x_0$ et $[] = 0$.
Supposons le résultat acquis au rang k, et prouvons-le au rang $k+1$: nous avons $r_{k+1} = 10r_k + i$, i étant le plus grand entier vérifiant $(20r_k + i)\times i \leqslant 100x_k + t_{k+1}$. Ainsi,

$$r_{k+1}^2 = 100r_k^2 + (20r_k + i)\times i \leqslant 100r_k^2 + 100x_k + t_{k+1} = 100[t_1 t_2 \cdots t_k] + t_{k+1} = [t_1 t_2 \cdots t_{k+1}]$$

Le caractère maximal de i prouve que $r_{k+1} = \left\lfloor\sqrt{[t_1 t_2 \cdots t_{k+1}]}\right\rfloor$. Enfin,

$$x_{k+1} = 100x_k + t_{k+1} - (20r_k + i)\times i = 100x_k + t_{k+1} - r_{k+1}^2 + 100r_k^2 = 100[t_1 t_2 \cdots t_k] + t_{k+1} - r_{k+1}^2 = [t_1 t_2 \cdots t_{k+1}] - r_{k+1}^2$$

ce qui achève la démonstration.

On en déduit que cette fonction retourne la valeur de $r_p = \left\lfloor \sqrt{[t_1 t_2 \cdots t_p]} \right\rfloor = \lfloor \sqrt{n} \rfloor$.

Pour obtenir les 1000 premières décimales de $\sqrt{2}$, il suffit d'appliquer cette fonction à l'entier 2×10^{2000} :

```
>>> sqrt(2*10**2000)
14142135623730950488016887242096980785696718753769480731766797379907324784621070 38...
```