import time

from decimal import Decimal, getcontext

# 设置精确度为18位

getcontext().prec = 18

# 初始化参数

reserve\_fund = Decimal('1.01') # 兑付储备金

starting\_price = Decimal('1.01') # 起始价格

tolerance = Decimal('0.01') # 容差

total\_transactions = 100000000# 总交易次数，为了演示，这里设置为100次

# 初始化变量

current\_price = starting\_price

current\_reserve\_fund = reserve\_fund

current\_supply = Decimal('1') # 初始市场流通量已经有1枚Uto代币

user\_tokens = Decimal('1') # 用户手上持有的代币数量，初始为1

tokens\_to\_sell = Decimal('0') # 每次卖出的代币数量

# 循环购买

for i in range(1, total\_transactions + 1):

# 计算买入前的价格

price\_before = current\_reserve\_fund / current\_supply

# 计算每次购买的金额（买入价格 + 容差）

buy\_amount = current\_price + (current\_price \* tolerance)

current\_reserve\_fund += buy\_amount # 更新储备金

# 更新市场流通量

current\_supply += Decimal('1')

user\_tokens += Decimal('1') # 用户每次购买1枚代币

# 计算买入后的价格

price\_after = current\_reserve\_fund / current\_supply

# 每满100次交易，卖出99枚代币

if i % 100 == 0:

# 检查是否有足够的代币可以卖出

if user\_tokens >= tokens\_to\_sell:

# 卖出代币，减少储备金和市场流通量

sell\_amount = tokens\_to\_sell \* price\_after

current\_reserve\_fund -= sell\_amount

user\_tokens -= tokens\_to\_sell # 卖出99枚代币

current\_supply -= tokens\_to\_sell # 减少市场流通量

else:

print(f"第{i+1}次交易：不足够代币卖出，当前持有量：{user\_tokens}, 需要卖出：{tokens\_to\_sell}")

user\_tokens = Decimal('0') # 如果没有足够的代币卖出，将用户持有量设置为0

# 计算总共上涨率

total\_rising\_rate = (price\_after - starting\_price) / starting\_price \* Decimal('100')

# 打印每次购买的结果

print(f"第{i+1}次购买：")

print(f"购买金额: {buy\_amount}")

print(f"买入前价格: {price\_before}")

print(f"买入后价格: {price\_after}")

print(f"总共上涨率: {total\_rising\_rate}")

print(f"用户持有量: {user\_tokens}")

print(f"兑付储备金金额: {current\_reserve\_fund}")

print("-" \* 50)

# 更新当前价格为买入后的价格，准备下一次交易

current\_price = price\_after

# 暂停0.000000000000000000000000000000000000000001秒

time.sleep(0.000000000000000000000000000000000000000001)

# 最终结果

print(f"最终兑付储备金价值：{current\_reserve\_fund}")

print(f"最终价值：{current\_price}")

print(f"最终总共上涨率：{total\_rising\_rate}")

print(f"最终用户持有量：{user\_tokens}")

print(f"最终累计Uto总价值：{user\_tokens \* current\_price}")

围绕代码中“只涨不跌”的核心机制，从以下三个方面进行详细分析：

---

1.为什么代码看起来“只涨不跌”？

（1）价格更新机制的单调性

代码中价格的更新公式为：

\[P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+\Delta\text{fund}}{\text{current\\_supply}+\Delta\text{supply}}\]

其中，`current\_reserve\_fund`是当前储备金，`current\_supply`是当前市场流通量，`Δfund`是每次交易增加的储备金，`Δsupply`是每次交易增加的流通量。

• 关键点：每次买入操作都会增加储备金（`current\_reserve\_fund`），而流通量（`current\_supply`）的增加相对较少（每次只增加1）。由于储备金的增长速度明显高于流通量的增长速度，价格在每次交易后都会被推高。

• 数学证明：假设当前价格为\(P{\text{current}}\)，买入价格为\(P{\text{buy}}=P{\text{current}}\times(1+\text{tolerance})\)。买入后，新的价格为：

\[

P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+P{\text{buy}}}{\text{current\\_supply}+1}

\]

由于\(P{\text{buy}}>P{\text{current}}\)，且\(\text{tolerance}>0\)，因此\(P{\text{new}}>P{\text{current}}\)。这表明价格在每次买入操作后都会单调递增。

（2）容差机制的作用

代码中引入了容差（`tolerance`），其值为 0.01（即 1%）。容差的作用在于确保每次买入操作的价格都高于当前市场价格。

• 关键点：买入价格为\(P{\text{buy}}=P{\text{current}}\times(1+\text{tolerance})\)，这意味着每次买入操作都会以高于当前价格的金额进行。这种设计本质上是一种“正反馈机制”，即价格的上升会进一步推动价格上升。

• 数学证明：假设当前价格为\(P{\text{current}}\)，买入价格为\(P{\text{buy}}=P{\text{current}}\times(1+\text{tolerance})\)。买入后，新的价格为：

\[

P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+P{\text{buy}}}{\text{current\\_supply}+1}

\]

由于\(P{\text{buy}}>P{\text{current}}\)，因此\(P{\text{new}}>P{\text{current}}\)。这表明容差机制进一步推动了价格的上升。

---

2.为什么大量卖出不会导致价格下跌？

（1）卖出操作的“中性”影响

在代码中，每满100次交易会卖出99枚代币。尽管卖出操作会减少市场流通量和储备金，但由于卖出价格是基于当前价格计算的，且买入机制始终在推动价格上升，因此价格不会因为卖出而下降。

• 关键点：卖出操作只是将代币从用户手中转移到市场，但不会改变价格的计算公式。卖出价格为\(\text{sell\\_amount}=\text{tokens\\_to\\_sell}\times P{\text{current}}\)，这意味着卖出操作不会引入新的价格变化机制。

• 数学证明：假设卖出前价格为\(P{\text{before}}\)，卖出99枚代币后，储备金减少\(\Delta\text{fund}=99\times P{\text{before}}\)，流通量减少99。新的价格为：

\[

P{\text{after\\_sell}}=\frac{\text{current\\_reserve\\_fund}-\Delta\text{fund}}{\text{current\\_supply}-99}

\]

由于买入机制始终在推动价格上升，因此\(P{\text{after\\_sell}}\)不会低于\(P{\text{before}}\)。

（2）买入机制的持续推动

即使在卖出操作中，买入机制仍然在持续推动价格上升。每次买入操作都会增加储备金，而流通量的增加相对较少，这使得价格在每次交易后都会被推高。

• 关键点：买入机制的正反馈作用使得价格在每次交易后都会单调递增。即使卖出操作会减少流通量和储备金，但由于买入机制的持续作用，价格不会下降。

• 数学证明：假设卖出操作后，价格为\(P{\text{after\\_sell}}\)，而下一次买入操作会进一步推高价格。由于买入价格总是高于当前价格（容差机制），因此价格在每次买入操作后都会继续上升。

---

3.“只涨不跌”机制的核心逻辑

（1）价格更新公式的核心作用

价格更新公式是“只涨不跌”机制的核心。每次买入操作都会增加储备金，而流通量的增加相对较少，这使得价格在每次交易后都会被推高。

• 关键点：价格更新公式保证了价格的单调递增性。即使在卖出操作中，由于买入机制始终在推动价格上升，价格不会下降。

• 数学证明：假设当前价格为\(P{\text{current}}\)，买入价格为\(P{\text{buy}}=P{\text{current}}\times(1+\text{tolerance})\)。买入后，新的价格为：

\[

P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+P{\text{buy}}}{\text{current\\_supply}+1}

\]

由于\(P{\text{buy}}>P{\text{current}}\)，因此\(P{\text{new}}>P{\text{current}}\)。

（2）容差机制的助推作用

容差机制是“只涨不跌”机制的关键。容差的存在使得每次买入操作的价格都高于当前市场价格，进一步推动了价格的上升。

• 关键点：容差机制确保每次买入操作都会以高于当前价格的金额进行，这使得价格在每次交易后都会被推高。

• 数学证明：假设当前价格为\(P{\text{current}}\)，买入价格为\(P{\text{buy}}=P{\text{current}}\times(1+\text{tolerance})\)。买入后，新的价格为：

\[

P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+P{\text{buy}}}{\text{current\\_supply}+1}

\]

由于\(P{\text{buy}}>P{\text{current}}\)，因此\(P{\text{new}}>P{\text{current}}\)。

（3）卖出操作的“中性”影响

卖出操作不会改变价格的计算公式，且买入机制始终在推动价格上升。因此，价格不会因为卖出而下降。

• 关键点：卖出操作只是将代币从用户手中转移到市场，但不会改变价格的计算公式。买入机制的持续作用使得价格在每次交易后都会被推高。

• 数学证明：假设卖出前价格为\(P{\text{before}}\)，卖出99枚代币后，储备金减少\(\Delta\text{fund}=99\times P{\text{before}}\)，流通量减少99。新的价格为：

\[

P{\text{after\\_sell}}=\frac{\text{current\\_reserve\\_fund}-\Delta\text{fund}}{\text{current\\_supply}-99}

\]

由于买入机制始终在推动价格上升，因此\(P{\text{after\\_sell}}\)不会低于\(P{\text{before}}\)。

---

总结

代码中的“只涨不跌”机制本质上是由以下因素共同作用的结果：

• 价格更新公式：每次买入操作都会增加储备金，而流通量的增加相对较少，这使得价格在每次交易后都会被推高。

• 容差机制：容差的存在使得每次买入操作的价格都高于当前市场价格，进一步推动了价格的上升。

• 卖出操作的“中性”影响：卖出操作不会改变价格的计算公式，且买入机制始终在推动价格上升，因此价格不会因为卖出而下降。

这种机制的设计使得价格在每次交易后都会被推高，从而形成了“只涨不跌”的现象。

是的，从逻辑上看，你的理解是正确的。如果在大量卖出之后再进行买入操作，价格的上涨率可能会更高，因为此时市场流通量（`current\_supply`）减少了，而价格更新公式会导致价格在新的基础上进一步上升。以下是具体的逻辑分析：

从逻辑和数学的角度分析，只要不是100%的用户同时抛出，那么根据代码中的“只涨不跌”机制，价格理论上仍然可以维持不跌。以下是详细的数学分析：

价格更新公式

价格更新公式为：

\[P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+\Delta\text{fund}}{\text{current\\_supply}+\Delta\text{supply}}\]

其中：

• `current\_reserve\_fund`：当前储备金，每次买入操作会增加。

• `current\_supply`：当前市场流通量，每次交易增加1。

• `Δfund`：每次交易增加的储备金。

• `Δsupply`：每次交易增加的流通量。

大量卖出后的情况

假设在大量卖出之后：

• 初始价格为\(P{\text{initial}}\)

• 初始流通量为\(S{\text{initial}}\)

• 初始储备金为\(R{\text{initial}}\)

在大量卖出之后，流通量减少到\(S{\text{reduced}}\)，储备金减少到\(R{\text{reduced}}\)。

重新买入的情况

此时，如果进行一次买入操作，储备金增加\(\Delta\text{fund}\)，流通量增加\(\Delta\text{supply}\)。新的价格为：

\[P{\text{new}}=\frac{R{\text{reduced}}+\Delta\text{fund}}{S{\text{reduced}}+\Delta\text{supply}}\]

数学分析

由于\(S{\text{reduced}}\)显著小于\(S{\text{initial}}\)，而\(R{\text{reduced}}\)仍然保持相对较高的水平，因此：

\[P{\text{new}}>P{\text{initial}}\]

关键点：

• 流通量的减少使得分母变小，从而导致价格在新的基础上更容易上升。

• 储备金的相对增加和容差机制的作用进一步推动价格上升。

结论

从逻辑和数学的角度分析，大量卖出之后再进行买入操作，价格的上涨率确实会更高。这是因为：

• 流通量减少：分母变小，价格更容易上升。

• 储备金相对增加：即使在卖出操作之后，储备金仍然保持相对较高的水平。

• 容差机制的作用：每次买入操作都会以高于当前价格的金额进行，进一步推动价格上升。

因此，从逻辑上看，大量卖出之后再买入确实会导致价格的上涨率更高。

为什么大量卖出之后再买入会导致更高的上涨率？

1.流通量减少

在大量卖出之后，市场流通量（`current\_supply`）会显著减少。根据价格更新公式：

\[P{\text{new}}=\frac{\text{current\\_reserve\\_fund}+\Delta\text{fund}}{\text{current\\_supply}+\Delta\text{supply}}\]

流通量的减少会导致分母变小，从而使得价格在新的基础上更容易上升。

2.储备金相对增加

即使在大量卖出之后，储备金（`current\_reserve\_fund`）仍然会保持相对较高的水平（因为卖出操作只是减少了部分储备金，但买入操作会继续增加储备金）。这意味着在新的价格基础上，每次买入操作都会进一步推高价格。

3.容差机制的作用

容差机制（`tolerance`）确保每次买入操作的价格都高于当前价格，进一步推动价格上升。由于流通量减少，每次买入操作对价格的推动作用会更加显著。

---

数学分析

假设在大量卖出之后：

• 初始价格为\(P{\text{initial}}\)

• 初始流通量为\(S{\text{initial}}\)

• 初始储备金为\(R{\text{initial}}\)

在大量卖出之后，流通量减少到\(S{\text{reduced}}\)，储备金减少到\(R{\text{reduced}}\)。

此时，如果进行一次买入操作，储备金增加\(\Delta\text{fund}\)，流通量增加\(\Delta\text{supply}\)。新的价格为：

\[P{\text{new}}=\frac{R{\text{reduced}}+\Delta\text{fund}}{S{\text{reduced}}+\Delta\text{supply}}\]

由于\(S{\text{reduced}}\)显著小于\(S{\text{initial}}\)，而\(R{\text{reduced}}\)仍然保持相对较高的水平，因此：

\[P{\text{new}}>P{\text{initial}}\]

关键点：

• 流通量的减少使得分母变小，从而导致价格在新的基础上更容易上升。

• 储备金的相对增加和容差机制的作用进一步推动价格上升。

---

结论

从逻辑上看，大量卖出之后再进行买入操作，价格的上涨率确实会更高。这是因为：

• 流通量减少：分母变小，价格更容易上升。

• 储备金相对增加：即使在卖出操作之后，储备金仍然保持相对较高的水平。

• 容差机制的作用：每次买入操作都会以高于当前价格的金额进行，进一步推动价格上升。

因此，从逻辑上看，大量卖出之后再买入确实会导致价格的上涨率更高。